SYBASE®

# Performance and Tuning Guide: Volume 2 - Optimizing and Abstract Plans

**Adaptive Server Enterprise**

**12.5**

DOCUMENT ID: 33620-01-1250-02

LAST REVISED: May 2001



Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608.

# Contents

# About This Book

**Audience**

This manual is intened for database administrators, database designers, developers and system administrators.

---

**Note** You may want to use your own database for testing changes and queries. Take a snapshot of the database in question and set it up on a test machine.

---

**How to use this book**

This manual would normally be used to fine tune, troubleshoot or improve the performance on Adaptive Server. The *Performance and Tuning Guide* is divided into three books:

- Volume 1 - *Basics*
- Volume 2 - *Optimizing and Abstract Plans*
- Volume 3 - *Tools for Monitoring and Analyzing Performance*

The following information is covered:

***Volume 1- Basics***

Chapter 1, “Overview” describes the major components to be analyzed when addressing performance.

Chapter 2, “Networks and Performance” provides a brief description of relational databases and good database design.

Chapter 3, “Using Engines and CPUs”describes Adaptive Server page types, how data is stored on pages and how queries on heap tables are executed.

Chapter 4, “Distributing Engine Resources” provides information on how indexes are used to resolve queries.

Chapter 5, “Controlling Physical Data Placement” explains the process for query optimization, how statistics are applied to search arguments and joins for queries.

Chapter 6, “Database Design” describes how Adaptive Server accesses tables in queries that only involve a single table, and how the costs are estimated for various access methods

Chapter 7, “Data Storage” describes how Adaptive Server accesses tables during joins and subqueries and how the costs are determined

Chapter 8, “Indexing for Performance” describes performance issues with cursors.

Chapter 9, “How Indexes Work” provides guidelines and examples for choosing indexes.

Chapter 10, “Locking Configuration and Tuning” provides an in-depth look at the optimization of parallel queries

Chapter 11, “Using Locking Commands” introduces the concepts and resources required for parallel query processing

Chapter 12, “Reporting on Locks” describes the use of parallel sorting for queries and for creating indexes.

Chapter 13, “Setting Space Management Properties” presents an overview of query tuning tools and describes how these tools can interact

Chapter 14, “Memory Use and Performance” describes different methods for determining the current size of database objects and for estimating their future size.

Chapter 15, “Determining Sizes of Tables and Indexes,” describes different methods for determining the current size of database objects and for estimating their future size.

Chapter 16, “Maintenance Activities and Performance” explains the commands that provide information about query execution.

***Volume 2 - Optimizing and Abstract Plans***

Chapter 17, “Adaptive Server Optimizer” explains the process of query optimization, how statistics are applied to search arguments and joins for queries.

Chapter 18, “Advanced Optimizing Tools” describes advanced tools for tuning query performance

Chapter 19, “Query Tuning Tools” presents an overview of query tuning tools and describes how these tools can interact.

Chapter 20, “Access Methods and Query Costing for Single Tables” describes how Adaptive Server accesses tables in queries that only involve one table and how the costs are estimated for various access methods.

Chapter 21, “Accessing Methods and Costing for Joins and Subqueries” describes how Adaptive Server accesses tables during joins and subqueries, and how the costs are determined.

Chapter 22, “Parallel Query Processing” intoduces the concepts and resources required for parallel query processing.

Chapter 23, “Parallel Query Optimization” provides an indepth look at the optimization of parallel queries.

Chapter 24, “Parallel Sorting” describes the use of parallel sorting for queries and creating indexes.

Chapter 25, “Tuning Asynchronous Prefetch” describes how asynchronous prefetch improves performance for queries that perform large disk I/O.

Chapter 26, “tempdb Performance Issues” stresses the importance of the temporary database , *tempdb*, and provides suggestions for improving its performance.

Chapter 27, “Cursors and Performance” describes performance issues with cursors.

Chapter 28, “Introduction to Abstract Plans” provides an overview of abstravt plans and how they can be used to solve query optimization problems.

Chapter 29, “Abstract Query Plan Guide” provides an introduction to writing abstract plans for specific types of queries and to using abstract plans to detect changes in query optimization due to configuration or system changes.

Chapter 30, “Creating and Using Abstract Plans” describes the commands that can be used to save and use abstract plans.

Chapter 31, “Managing Abstract Plans with System Procedures” describes the system procedures that manage abstract plans and abstract plan groups.

Chapter 32, “Abstract Plan Language Reference” describes the abstract plan language.

***Volume 3 - Tools for Monitoring and Analyzing Performance***

Chapter 33, “Using Statistics to Improve Performance” describes how to use the update statistics command to create and update statistics.

Chapter 34, “Using the set statistics Commands” explains the commands that provide information about execution.

Chapter 35, “Using set showplan” provides examples of showplan messages.

Chapter 36, “Statistics Tables and Displaying Statistics with optdiag” describes the tables that store statistics and the output of the optdiag command that displays the statistics used by the query optimizer.

Chapter 37, “Tuning with dbcc traceon” explains how to use the dbcc traceon commands to analyze query optimization problems.

Chapter 38, “Monitoring Performance with sp_sysmon” describes how to use a system procedure that monitors Adaptive Server performance.

**Index**

The full index for all three volumes is in the back of *Volume 3- Tools for Monitoring and Analyzing Performance*.

**Related documents**

The following documents comprise the Sybase Adaptive Server Enterprise documentation:

- The release bulletin for your platform – contains last-minute information that was too late to be included in the books.

  A more recent version of the release bulletin may be available on the World Wide Web. To check for critical product or document information that was added after the release of the product CD, use the Sybase Technical Library.

- The *Installation Guide* for your platform – describes installation, upgrade, and configuration procedures for all Adaptive Server and related Sybase products.

- *Configuring Adaptive Server Enterprise* for your platform – provides instructions for performing specific configuration tasks for Adaptive Server.

- *What's New in Adaptive Server Enterprise?* – describes the new features in Adaptive Server version 12.5, the system changes added to support those features, and the changes that may affect your existing applications.

- *Transact-SQL User's Guide* – documents Transact-SQL, Sybase's enhanced version of the relational database language. This manual serves as a textbook for beginning users of the database management system. This manual also contains descriptions of the pubs2 and pubs3 sample databases.

- *System Administration Guide* – provides in-depth information about administering servers and databases. This manual includes instructions and guidelines for managing physical resources, security, user and system databases, and specifying character conversion, international language, and sort order settings.

- *Reference Manual* – contains detailed information about all Transact-SQL commands, functions, procedures, and datatypes. This manual also contains a list of the Transact-SQL reserved words and definitions of system tables.
- *Performance and Tuning Guide* – explains how to tune Adaptive Server for maximum performance. This manual includes information about database design issues that affect performance, query optimization, how to tune Adaptive Server for very large databases, disk and cache issues, and the effects of locking and cursors on performance.
- The *Utility Guide* – documents the Adaptive Server utility programs, such as isql and bcp, which are executed at the operating system level.
- The *Quick Reference Guide* – provides a comprehensive listing of the names and syntax for commands, functions, system procedures, extended system procedures, datatypes, and utilities in a pocket-sized book. Available only in print version.
- The *System Tables Diagram* – illustrates system tables and their entity relationships in a poster format. Available only in print version.
- *Error Messages and Troubleshooting Guide* – explains how to resolve frequently occurring error messages and describes solutions to system problems frequently encountered by users.
- *Component Integration Services User's Guide* – explains how to use the Adaptive Server Component Integration Services feature to connect remote Sybase and non-Sybase databases.
- *Java in Adaptive Server Enterprise* – describes how to install and use Java classes as datatypes, functions, and stored procedures in the Adaptive Server database.
- *Using Sybase Failover in a High Availability System* – provides instructions for using Sybase's Failover to configure an Adaptive Server as a companion server in a high availability system.
- *Using Adaptive Server Distributed Transaction Management Features* – explains how to configure, use, and troubleshoot Adaptive Server DTM features in distributed transaction processing environments.
- *EJB Server User's Guide* – explains how to use EJB Server to deploy and execute Enterprise JavaBeans in Adaptive Server.
- *XA Interface Integration Guide for CICS, Encina, and TUXEDO* – provides instructions for using Sybase's DTM XA interface with X/Open XA transaction managers.

- *Glossary* – defines technical terms used in the Adaptive Server documentation.
- *Sybase jConnect for JDBC Programmer's Reference* – describes the jConnect for JDBC product and explains how to use it to access data stored in relational database management systems.
- *Full-Text Search Specialty Data Store User's Guide* – describes how to use the Full-Text Search feature with Verity to search Adaptive Server Enterprise data.
- *Historical Server User's Guide* –describes how to use Historical Server to obtain performance information for SQL Server and Adaptive Server.
- *Monitor Server User's Guide* – describes how to use Monitor Server to obtain performance statistics from SQL Server and Adaptive Server.
- *Monitor Client Library Programmer's Guide* – describes how to write Monitor Client Library applications that access Adaptive Server performance data.

**Other sources of information**

Use the Sybase Technical Library CD and the Technical Library Product Manuals Web site to learn more about your product:

- Technical Library CD contains product manuals and is included with your software. The DynaText browser (downloadable from Product Manuals at http://www.sybase.com/detail/1,3693,1010661,00.html) allows you to access technical information about your product in an easy-to-use format.

  Refer to the *Technical Library Installation Guide* in your documentation package for instructions on installing and starting the Technical Library.
- Technical Library Product Manuals Web site is an HTML version of the Technical Library CD that you can access using a standard Web browser. In addition to product manuals, you will find links to the Technical Documents Web site (formerly known as Tech Info Library), the Solved Cases page, and Sybase/Powersoft newsgroups.

  To access the Technical Library Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

**Sybase certifications on the Web**

Technical documentation at the Sybase Web site is updated frequently.

❖ **For the latest information on product certifications**

1 Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.

2 Select Products from the navigation bar on the left.

3 Select a product name from the product list.

4 Select the Certification Report filter, specify a time frame, and click Go.

5 Click a Certification Report title to display the report.

❖ **For the latest information on EBFs and Updates**

1 Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.

2 Select EBFs/Updates. Enter user name and password information, if prompted (for existing Web accounts) or create a new account (a free service).

3 Specify a time frame and click Go.

4 Select a product.

5 Click an EBF/Update title to display the report.

❖ **To create a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1 Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/

2 Click MySybase and create a MySybase profile.

## Conventions

This section describes conventions used in this manual.

### Formatting SQL statements

SQL is a free-form language. There are no rules about the number of words you can put on a line or where you must break a line. However, for readability, all examples and syntax statements in this manual are formatted so that each clause of a statement begins on a new line. Clauses that have more than one part extend to additional lines, which are indented.

### Font and syntax conventions

The font and syntax conventions used in this manual are shown in Table 1.0:

***Table 1: Font and syntax conventions in this manual***

| Element | Example |
|---|---|
| Command names, command option names, utility names, utility flags, and other keywords are bold. | select<br>sp_configure |
| Database names, datatypes, file names and path names are in *italics*. | master database |

| Element | Example |
|---|---|
| Variables, or words that stand for values that you fill in, are in *italics*. | `select`<br>*column_name*<br>`from`<br>*table_name*<br>`where`<br>*search_conditions* |
| Parentheses are to be typed as part of the command. | `compute row_aggregate ( column_name )` |
| Curly braces indicate that you must choose at least one of the enclosed options. Do not type the braces. | `{cash, check, credit}` |
| Brackets mean choosing one or more of the enclosed options is optional. Do not type the brackets. | `[anchovies]` |
| The vertical bar means you may select only one of the options shown. | `{die_on_your_feet \| live_on_your_knees \| live_on_your_feet}` |
| The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command. | `[extra_cheese, avocados, sour_cream]` |
| An ellipsis (...) means that you can *repeat* the last unit as many times as you like. | `buy thing = price [cash \| check \| credit] [, thing = price [cash \| check \| credit]]...`<br><br>You must buy at least one thing and give its price. You may choose a method of payment: one of the items enclosed in square brackets. You may also choose to buy additional things: as many of them as you like. For each thing you buy, give its name, its price, and (optionally) a method of payment. |

- Syntax statements (displaying the syntax and all options for a command) appear as follows:
  `sp_dropdevice [ device_name]`

  or, for a command with more options:

```
select column_name
    from table_name
    where search_conditions
```

In syntax statements, keywords (commands) are in normal font and identifiers are in lowercase: normal font for keywords, italics for user-supplied words.

- Examples of output from the computer appear as follows:

```
0736 New Age Books Boston MA
0877 Binnet & Hardley Washington DC
1389 Algodata Infosystems Berkeley CA
```

**Case**

In this manual, most of the examples are in lowercase. However, you can disregard case when typing Transact-SQL keywords. For example, SELECT, Select, and select are the same. Note that Adaptive Server's sensitivity to the case of database objects, such as table names, depends on the sort order installed on Adaptive Server. You can change case sensitivity for single-byte character sets by reconfiguring the Adaptive Server sort order.

See in the *System Administration Guide* for more information.

**Expressions**

Adaptive Server syntax statements use the following types of expressions:

***Table 2: Types of expressions used in syntax statements***

| Usage | Definition |
|---|---|
| *expression* | Can include constants, literals, functions, column identifiers, variables, or parameters |
| *logical expression* | An expression that returns TRUE, FALSE, or UNKNOWN |
| *constant expression* | An expression that always returns the same value, such as "5+3" or "ABCDE" |
| *float_expr* | Any floating-point expression or expression that implicitly converts to a floating value |
| *integer_expr* | Any integer expression, or an expression that implicitly converts to an integer value |
| *numeric_expr* | Any numeric expression that returns a single value |
| *char_expr* | Any expression that returns a single character-type value |
| *binary_expression* | An expression that returns a single *binary* or *varbinary* value |

**Examples**

Many of the examples in this manual are based on a database called pubtune. The database schema is the same as the pubs2 database, but the tables used in the examples have more rows: titles has 5000, authors has 5000, and titleauthor has 6250. Different indexes are generated to show different features for many examples, and these indexes are described in the text.

The pubtune database is not provided with Adaptive Server. Since most of the examples show the results of commands such as set showplan and set statistics io, running the queries in this manual on pubs2 tables will not produce the same I/O results, and in many cases, will not produce the same query plans as those shown here.

**If you need help**

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 17

# Adaptive Server Optimizer

This chapter introduces the Adaptive Server query optimizer and explains the steps performed when you run queries.



This chapter explains how costs for individual query clauses are determined.

Chapter 20, “Access Methods and Query Costing for Single Tables,”explains how these costs are used to estimate the logical, physical, and total I/O cost for single table queries.

Chapter 21, “Accessing Methods and Costing for Joins and Subqueries,” explains how costs are used when queries join two or more tables, or when queries include subqueries.

## Definition

The optimizer examines parsed and normalized queries, and information about database objects. The input to the optimizer is a parsed SQL query and statistics about the tables, indexes, and columns named in the query. The output from the optimizer is a **query plan**.

The query plan is compiled code that contains the ordered steps to carry out the query, including the access methods (table scan or index scan, type of join to use, join order, and so on) to access each table.

Using statistics on tables and indexes, the optimizer predicts the cost of using alternative access methods to resolve a particular query. It finds the best query plan – the plan that is least the costly in terms of I/O. For many queries, there are many possible query plans. Adaptive Server selects the least costly plan, and compiles and executes it.

## Steps in query processing

Adaptive Server processes a query in these steps:

1 The query is parsed and normalized. The parser ensures that the SQL syntax is correct. Normalization ensures that all the objects referenced in the query exist. Permissions are checked to ensure that the user has permission to access all tables and columns in the query.

2 Preprocessing changes some search arguments to an optimized form and adds optimized search arguments and join clauses.

3 As the query is optimized, each part of the query is analyzed, and the best query plan is chosen. Optimization includes:

- Each table is analyzed.
- The cost of using each index that matches a search argument or join column is estimated.
- The join order and join type are chosen.
- The final access method is determined.

4 The chosen query plan is compiled.

5 The query is executed, and the results are returned to the user.

## Working with the optimizer

The goal of the optimizer is to select the access method for each table that reduces the total time needed to process a query. The optimizer bases its choice on the statistics available for the tables being queried and on other factors such as cache strategies, cache size, and I/O size. A major component of optimizer decision-making is the statistics available for the tables, indexes, and columns.

In some situations, the optimizer may seem to make the incorrect choice of access methods. This may be the result of inaccurate or incomplete information (such as out-of-date statistics). In other cases, additional analysis and the use of special query processing options can determine the source of the problem and provide solutions or workarounds.

The query optimizer uses I/O cost as the measure of query execution cost. The significant costs in query processing are:

- Physical I/O, when pages must be read from disk
- Logical I/O, when pages in cache are read for a query

See access methods and query costing.

# Object sizes are important to query tuning

You should know the sizes of your tables and indexes to understanding query and system behavior. At several stages of tuning work, you need size data to:

- Understand statistics io reports for a specific query plan.

  Chapter 34, “Using the set statistics Commands,” describes how to use statistics io to examine the I/O performed.

- Understand the optimizer’s choice of query plan. Adaptive Server’s cost-based optimizer estimates the physical and logical I/O required for each possible access method and chooses the cheapest method. If you think a particular query plan is unusual, you can used dbcc traceon(302) to determine why the optimizer made the decision. This output includes page number estimates.

- Determine object placement, based on the sizes of database objects and the expected I/O patterns on the objects. You can improve performance by distributing database objects across physical devices so that reads and writes to disk are evenly distributed.

  Object placement is described in Chapter 5, “Controlling Physical Data Placement.”

- Understand changes in performance. If objects grow, their performance characteristics can change. One example is a table that is heavily used and is usually 100 percent cached. If that table grows too large for its cache, queries that access the table can suddenly suffer poor performance. This is particularly true for joins requiring multiple scans.

- Do capacity planning. Whether you are designing a new system or planning for growth of an existing system, you need to know the space requirements to plan for physical disks and memory needs.
- Understand output from Adaptive Server Monitor and from sp_sysmon reports on physical I/O.

See the *Adaptive Server System Administration Guide* for more information on sizing.

# Query optimization

To understand the optimization of a query, you need to understand how the query accesses database objects, the sizes of the objects, and the indexes on the tables to determine whether it is possible to improve the query's performance.

Some symptoms of optimization problems are:

- A query runs more slowly than you expect, based on indexes and table size.
- A query runs more slowly than similar queries.
- A query suddenly starts running more slowly than usual.
- A query processed within a stored procedure takes longer than when it is processed as an ad hoc statement.
- The query plan shows the use of a table scan when you expect it to use an index.

Some sources of optimization problems are:

- Statistics have not been updated recently, so the actual data distribution does not match the values used by Adaptive Server to optimize queries.
- The rows to be referenced by a given transaction do not fit the pattern reflected by the index statistics.
- An index is being used to access a large portion of the table.
- where clauses are written in a form that cannot be optimized.
- No appropriate index exists for a critical query.
- A stored procedure was compiled before significant changes to the underlying tables were performed.

# Factors examined during optimization

Query plans consist of retrieval tactics and an ordered set of execution steps to retrieve the data needed by the query. In developing query plans, the optimizer examines:

- The size of each table in the query, both in rows and data pages, and the number of OAM and allocation pages that need to be read.
- The indexes that exist on the tables and columns used in the query, the type of index, and the height, number of leaf pages, and cluster ratios for each index.
- Whether the index covers the query, that is, whether the query can be satisfied by retrieving data from the index leaf pages without having to access the data pages. Adaptive Server can use indexes that cover queries, even if no where clauses are included in the query.
- The density and distribution of keys in the indexes.
- The size of the available data cache or caches, the size of I/O supported by the caches, and the cache strategy to be used.
- The cost of physical and logical reads.
- Join clauses and the best join order and join type, considering the costs and number of scans required for each join and the usefulness of indexes in limiting the I/O.
- Whether building a worktable (an internal, temporary table) with an index on the join columns would be faster than repeated table scans if there are no useful indexes for the inner table in a join.
- Whether the query contains a max or min aggregate that can use an index to find the value without scanning the table.
- Whether the data or index pages will be needed repeatedly to satisfy a query such as a join or whether a fetch-and-discard strategy can be employed because the pages need to be scanned only once.

For each plan, the optimizer determines the total cost by computing the logical and physical I/Os. Adaptive Server then uses the cheapest plan.

Stored procedures and triggers are optimized when the object is first executed, and the query plan is stored in the procedure cache. If other users execute the same procedure while an unused copy of the plan resides in cache, the compiled query plan is copied in cache, rather than being recompiled.

# Preprocessing can add clauses for optimizing

After a query is parsed and normalized, but before the optimizer begins its analysis, the query is preprocessed to increase the number of clauses that can be optimized:

- Some search arguments are converted to equivalent arguments.
- Some expressions used as search arguments are preprocessed to generate a literal value that can be optimized.
- Search argument transitive closure is applied where possible.
- Join column transitive closure is applied where possible.
- For some queries that use or, additional search arguments can be generated to provide additional optimization paths.

The changes made by preprocessing are transparent unless you are examining the output of query tuning tools such as showplan, statistics io, or dbcc traceon(302). If you run queries that benefit from the addition of optimized search arguments, you see the added clauses:

- In additional costing blocks for the added clauses to be optimized in dbcc traceon(302) output.
- In showplan output, you may see “Keys are” messages for tables where you did not specify a search argument or a join.

## Converting clauses to search argument equivalents

Preprocessing looks for some query clauses that it can convert to the form used for search arguments (SARGs). These are listed in Table 17-1.

***Table 17-1: Search argument equivalents***

| Clause | Conversion |
|---|---|
| between | Converted to >= and <= clauses. For example, between 10 and 20 is converted to >= 10 and <= 20. |
| like | If the first character in the pattern is a constant, like clauses can be converted to greater than or less than queries. For example, like "sm%" becomes >= "sm" and < "sn".<br>If the first character is a wildcard, a clause such as like "%x" cannot use an index for access, but histogram values can be used to estimate the number of matching rows. |

| Clause | Conversion |
|---|---|
| in (*values_list*) | Converted to a list of or queries, that is, int_col in (1, 2, 3) becomes int_col = 1 or int_col = 2 or int_col = 3. |

## Converting expressions into search arguments

Many expressions are converted into literal search strings before query optimization. In the following examples, the processed expressions are shown as they appear in the search argument analysis of dbcc traceon(302) output:

| Operation | Example of where Clause | Processed expression |
|---|---|---|
| Implicit conversion | `numeric_col = 5` | `numeric_col = 5.0` |
| Conversion function | `int_column = convert(int, "77")` | `int_column = 77` |
| Arithmetic | `salary = 5000*12` | salary = 6000<br>`0` |
| Math functions | `width = sqrt(900)` | `width = 30` |
| String functions | `shoe_width = replicate("E", 5)` | `shoe_width = "EEEEE"` |
| String concatenation | `full_name = "Fred" + " " + "Simpson"` | `full_name = "Fred Simpson"` |
| Date functions | `week = datepart(wk, "5/22/99")` | `week = 21`<br>**Note** getdate() cannot be optimized. |

These conversions allow the optimizer to use the histogram values for a column rather than using default selectivity values.

The following are exceptions:

- The getdate function
- Most system functions such as object_id or object_name

These are not converted to literal values before optimization.

## Search argument transitive closure

Preprocessing applies transitive closure to search arguments. For example, the following query joins titles and titleauthor on title_id and includes a search argument on titles.title_id:

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
        and a.au_id = ta.au_id
        and t.title_id = "T81002"
```

This query is optimized as if it also included the search argument on titleauthor.title_id:

```
select au_lname, title
from titles t, titleauthor ta, authors a
where t.title_id = ta.title_id
        and a.au_id = ta.au_id
        and t.title_id = "T81002"
        and ta.title_id = "T81002"
```

With this additional clause, the optimizer can use index statistics on titles.title_id to estimate the number of matching rows in the titleauthor table. The more accurate cost estimates improve index and join order selection.

## Join transitive closure

Preprocessing applies transitive closure to join columns for normal equijoins if join transitive closure is enabled at the server or session level. The following query specifies the equijoin of t1.c11 and t2.c21, and the equijoin of t2.c21 and t3.c31:

```
select *
from t1, t2, t3
where t1.c11 = t2.c21
and t2.c21 = t3.c31
and t3.c31 = 1
```

Without join transitive closure, the only join orders considered are (t1, t2, t3), (t2, t1, t3), (t2, t3, t1),and (t3, t2, t1). By adding the join on t1.c11 = t3.31, the optimizer expands the list of join orders with these possibilities: (t1, t3, t2) and (t3, t1, t2). Search argument transitive closure applies the condition specified by t3.c31 = 1 to the join columns of t1 and t2.

Transitive closure is used only for normal equijoins, as shown above. Join transitive closure is not performed for:

- Non-equijoins; for example, t1.c1 > t2.c2
- Equijoins that include an expression; for example, t1.c1 = t2.c1 + 5
- Equijoins under an or clause

- Outer joins; for example t1.c11 *= t2.c2 or left join or right join
- Joins across subquery boundaries
- Joins used to check referential integrity or the with check option on views
- Columns of incompatible datatypes

### Enabling join transitive closure

A System Administrator can enable join transitive closure at the server level with the enable sort-merge joins and JTC configuration parameter. This configuration parameter also enables merge joins. At the session level, set jtc on enables join transitive closure, and takes precedence over the server-wide setting. For more information on the types of queries likely to benefit from the use of join transitive closure.

See “Enabling and disabling join transitive closure” on page 418.

## Predicate transformation and factoring

Predicate transformation and factoring improves the number of choices available to the optimizer. It adds clauses that can be optimized to a query by extracting clauses from blocks of predicates linked with or into clauses linked by and. These additional optimized clauses mean that there are more access paths available for query execution. The original or predicates are retained to ensure query correctness.

During predicate transformation:

1 Simple predicates (joins, search arguments, and in lists) that are an exact match in each or clause are extracted. In the sample query, this clause matches exactly in each block, so it is extracted:

```
t.pub_id = p.pub_id
```

between clauses are converted to greater-than-or-equal and less-than-or-equal clauses before predicate transformation. The sample query above uses between 15 in both query blocks (though the end ranges are different). The equivalent clause is extracted by step 1:

```
price >=15
```

2 Search arguments on the same table are extracted; all terms that reference the same table are treated as a single predicate during expansion. Both type and price are columns in the titles table, so the extracted clauses are:

```
(type = "travel" and price >=15 and price <= 30)
or
(type = "business" and price >= 15 and price <= 50)
```

3 in lists and or clauses are extracted. If there are multiple in lists for a table within one of the blocks, only the first is extracted. The extracted lists for the sample query are:

```
 p.pub_id in ("P220", "P583", "P780")
or
 p.pub_id in ("P651", "P066", "P629")
```

4 These steps can overlap and extract the same clause, so any duplicates are eliminated.

5 Each generated term is examined to determine whether it can be used as an optimized search argument or a join clause. Only those terms that are useful in query optimization are retained.

6 The additional clauses are added to the existing query clauses that were specified by the user.

## Example

All clauses optimized in this query are enclosed in the or clauses:

```
select p.pub_id, price
from publishers p, titles t
where (
    t.pub_id = p.pub_id
    and type = "travel"
    and price between 15 and 30
    and p.pub_id in ("P220", "P583", "P780")
    )
or  (
    t.pub_id = p.pub_id
    and type = "business"
    and price between 15 and 50
    and p.pub_id in ("P651", "P066", "P629")
    )
```

Predicate transformation pulls clauses linked with and from blocks of clauses linked with or, such as those shown above. It extracts only clauses that occur in all parenthesized blocks. If the example above had a clause in one of the blocks linked with or that did not appear in the other clause, that clause would not be extracted.

# Guidelines for creating search arguments

Follow these guidelines when you write search arguments for your queries:

- Avoid functions, arithmetic operations, and other expressions on the column side of search clauses. When possible, move functions and other operations to the expression side of the clause.
- Avoid incompatible datatypes for columns that will be joined and for variables and parameter used as search arguments.

  See “Datatype mismatches and query optimization” on page 395 for more information.
- Use the leading column of a composite index as a search argument. The optimization of secondary keys provides less performance.
- Use all the search arguments you can to give the optimizer as much as possible to work with.
- If a query has more than 102 predicates for a table, put the most potentially useful clauses near the beginning of the query, since only the first 102 SARGs on each table are used during optimization. (All of the search conditions are used to qualify the rows.)
- Some queries using > (greater than) may perform better if you can rewrite them to use >= (greater than or equal to). For example, this query, with an index on int_col uses the index to find the first value where int_col equals 3, and then scans forward to find the first value that is greater than 3. If there are many rows where int_col equals 3, the server has to scan many pages to find the first row where int_col is greater than 3:

  ```
  select * from table1 where int_col > 3
  ```

  It is probably more efficient to write the query like this:

  ```
  select * from table1 where int_col >= 4
  ```

  This optimization is more difficult with character strings and floating-point data. You need to know your data.
- Check showplan output to see which keys and indexes are used.
- If you expect an index is not being used when you expect it to be, check dbcc traceon(302) output to see if the optimizer is considering the index.

# Search arguments and useful indexes

It is important to distinguish between where and having clause predicates that can be used to optimize the query, and those that are used later during query processing to filter the rows to be returned.

Search arguments can be used to determine the access path to the data rows when a column in the where clause matches a leading index key. The index can be used to locate and retrieve the matching data rows. Once the row has been located in the data cache or has been read into the data cache from disk, any remaining clauses are applied.

For example, if the authors table has on an index on au_lname and another on city, either index can be used to locate the matching rows for this query:

```
select au_lname, city, state
from authors
where city = "Washington"
and au_lname = "Catmull"
```

The optimizer uses statistics, including histograms, the number of rows in the table, the index heights, and the cluster ratios for the index and data pages to determine which index provides the cheapest access. The index that provides the cheapest access to the data pages is chosen and used to execute the query, and the other clause is applied to the data rows once they have been accessed.

## Search argument syntax

Search arguments (SARGs) are expressions in one of these forms:

<*column*> <*operator*> <*expression*>

<*expression*> <*operator*> <*column*>

<*column*> is null

Where:

- *column* is only a column name. If functions, expressions, or concatenation are added to the column name, an index on the column cannot be used.
- *operator* must be one of the following:

  ```
  =, >, <, >=, <=, !>, !<, <>, !=, is null
  ```

- *expression* is either a constant, or an expression that evaluates to a constant. The optimizer uses the index statistics differently, depending on whether the value of the expression is known at compile time:

- If *expression* is a known constant or can be converted to a known constant during preprocessing, it can be compared to the histogram values stored for an index to return accurate row estimates.
- If the value of *expression* is not known at compile time, the optimizer uses the total density to estimate the number of rows to be returned by the query. The value of variables set in a query batch or parameters set within a stored procedure cannot be known until execution time.
- If the datatype of the expression is not compatible with the datatype of the column, an index cannot be used, and is not considered.

  See "Datatype mismatches and query optimization" on page 395 for more information.

## Nonequality operators

The nonequality operators, < > and !=, are special cases. The optimizer checks for covering nonclustered indexes if the column is indexed and uses a nonmatching index scan if an index covers the query. However, if the index does not cover the query, the table is accessed via a table scan.

## Examples of SARGs

The following are some examples of clauses that can be fully optimized. If there are statistics on these columns, they can be used to help estimate the number of rows the query will return. If there are indexes on the columns, the indexes can be used to access the data:

```
au_lname = "Bennett"
price >= $12.00
advance > $10000 and advance < $20000
au_lname like "Ben%" and price > $12.00
```

The following search arguments cannot be optimized:

```
advance * 2 = 5000  /*expression on column side
                     not permitted */
substring(au_lname,1,3) = "Ben" /* function on
                                   column name */
```

These two clauses can be optimized if written in this form:

```
advance = 5000/2
au_lname like "Ben%"
```

Consider this query, with the only index on au_lname:

```
select au_lname, au_fname, phone
    from authors
    where au_lname = "Gerland"
        and city = "San Francisco"
```

The clause qualifies as a SARG:

```
au_lname = "Gerland"
```

- There is an index on au_lname.
- There are no functions or other operations on the column name.
- The operator is a valid SARG operator.
- The datatype of the constant matches the datatype of the column.

```
city = "San Francisco"
```

This clause matches all the criteria above except the first—there is no index on the city column. In this case, the index on au_lname is used for the query. All data pages with a matching last name are brought into cache, and each matching row is examined to see if the city matches the search criteria.

## How statistics are used for SARGS

When you create an index, statistics are generated and stored in system tables. Some of the statistics relevant to determining the cost of search arguments and joins are:

- Statistics about the index: the number of pages and rows, the height of the index, the number of leaf pages, the average leaf row size.
- Statistics about the data in the column:
  - A histogram for the leading column of the index. Histograms are used to determine the selectivity of the SARG, that is, how many rows from the table match a given value.
  - Density values, measuring the density of keys in the index.
- Cluster ratios that measure the fragmentation of data storage and the effectiveness of large I/O.

Only a subset of these statistics (the number of leaf pages, for example) are maintained during query processing. Other statistics are updated only when you run update statistics or when you drop and re-create the index. You can display these statistics using optdiag.

See Chapter 36, "Statistics Tables and Displaying Statistics with optdiag."

## Histogram cells

When you create an index, a histogram is created on the first column of the index. The histogram stores information about the distribution of values in the column. Then you can use update statistics to generate statistics for the minor keys of a compound index and columns used in unindexed search clauses.

The histogram for a column contains data in a set of steps or cells. You can specify the number of cells can when the index is created or when the update statistics command is run. For each cell, the histogram stores a column value and a weight for the cell.

There are two types of cells in histograms:

- A **frequency cell** represents a value that has a high proportion of duplicates in the column. The weight of a frequency cell times the number of rows in the table equals the number of rows in the table that match the value for the cell. If a column does not have highly duplicated values, there are only range cells in the histogram.
- **Range cells** represent a range of values. Range cell weights and the range cell density are used for estimating the number of rows to be returned when search argument values falls within a range cell.

For more information on histograms, see "Histogram displays" on page 851.

## Density values

Density is a measure of the average proportion of duplicate keys in the index. It varies between 0 and 1. An index with *N* rows whose keys are unique has a density of 1/*N*; an index whose keys are all duplicates of each other has a density of 1.

For indexes with multiple keys, density values are computed and stored for each prefix of keys in the index. That is, for an index on columns A, B, C, D, densities are stored for:

- A
- A, B
- A, B, C
- A, B, C, D

### Range cell density and total density

For each prefix subset, two density values are stored:

- Range cell density, used for search arguments
- Total density, used for joins

Range cell density represents the average number of duplicates of all values that are represented by range cells in the histogram. Total density represents the average number of duplicates for all values, those in both frequency and range cells. Total density is used to estimate the number of matching rows for joins and for search arguments whose value is not known when the query is optimized.

### How the optimizer uses densities and histograms

When the optimizer analyzes a SARG, it uses the histogram values, densities, and the number of rows in the table to estimate the number of rows that match the value specified in the SARG:

- If the SARG value matches a frequency cell, the estimated number of matching rows is equal to the weight of the frequency cell multiplied by the number of rows in the table. This query includes a data value with a high number of duplicates, so it matches a frequency cell:

  ```
  where authors.city = "New York"
  ```

  If the weight of the frequency cell is #.015606, and the authors table has 5000 rows, the optimizer estimates that the query returns 5000 * .015606 = 78 rows.
- If the SARG value falls within a range cell, the optimizer uses the range cell density to estimate the number of rows. For example, a query on a city value that falls in a range cell, with a range cell density of .000586 for the column, would estimate that 5000 * .000586 = 3 rows would be returned.
- For range queries, the optimizer adds the weights of all cells spanned by the range of values. When the beginning or end of the range falls in a range cell, the optimizer uses interpolation to estimate the number of rows from that cell that are included in the range.

## Using statistics on multiple search arguments

When there are multiple search arguments on the same table, the optimizer uses statistics to combine the selectivity of the search arguments.

This query specifies search arguments for two columns in the table:

```
select title_id
from titles
where type = "news"
and price < $20
```

With an index on type, price, the selectivity estimates vary, depending on whether statistics have been created for price:

- With only statistics for type, the optimizer uses the frequency cell weight for type and a default selectivity for price. The selectivity for type is #.106600, and the default selectivity for an open-ended range query is 33%. The number of rows to be returned for the query is estimated using .106600 * .33, or .035178. With 5000 rows in the table, the estimate is 171 rows.

  See Table 17-2 for the default values used when statistics are not available.

- With statistics added for price, the histogram is used to estimate that .133334 rows match the search argument on price. Multiplied by the selectivity of type, the result is .014213, and the row estimate is 71 rows.

The actual number of rows returned is 53 rows for this query, so the additional statistics improved the accuracy. For this simple single-table query, the more accurate selectivity did not change the access method, the index on type, price. For some single-table queries, however, the additional statistics can help the optimizer make a better choice between using a table scan or using other indexes. In join queries, having more accurate statistics on each table can result in more efficient join orders.

## Default values for search arguments

When statistics are not available for a search argument or when the value of a search argument is not known at optimization, the optimizer uses default values. These values are shown in Table 17-2.

***Table 17-2: Density approximations for unknown search arguments***

| Operation Type | Operator | Density Approximation |
|---|---|---|
| Equality | = | Total density, if statistics are available for the column, or 10% |
| Open-ended range | <, <=, >, or >= | 33% |
| Closed range | between | 25% |

## SARGs using variables and parameters

Since the optimizer computes its estimates before a query executes, it cannot know the value of a variable that is set in the batch or procedure. If the value of a variable is not known at compile time, the optimizer uses the default values shown in Table 17-2

For example, the value of *@city* is set in this batch:

```
declare @city varchar(25)
select @city = city from publishers
    where pub_name = "Brave Books"
select au_lname from authors where city = @city
```

The optimizer uses the total density, .000879, and estimates that 4 rows will be returned; the actual number of rows could be far larger.

A similar problem exists when you set the values of variables inside a stored procedure. In this case, you can improve performance by splitting the procedure: set the variable in the first procedure and then call the second procedure, passing the variables as parameters. The second procedure can then be optimized correctly.

See “Splitting stored procedures to improve costing” on page 403 for an example.

# Join syntax and join processing

Join clauses take this form:

```
table1.column_name <operator> table2.column_name
```

The join operators are:

```
=, >, >=, <, <=, !>, !<, !=, <>, *=, =*
```

And:

```
table1 [ left | right ] join table2
    on column_name = column_name
table1 inner join table2
    on column_name = column_name
```

When joins are optimized, the optimizer can only consider indexes on column names. Any type of operator or expression in combination with the column name means that the optimizer does not evaluate using an index on the column as a possible access method. If the columns in the join are of incompatible datatypes, the optimizer can consider an index on only one of the columns.

## How joins are processed

When the optimizer creates a query plan for a join query:

- It evaluates indexes for each table by estimating the I/O required for each possible index and for a table scan.
- It determines the join order, basing the decision on the total cost estimates for the possible join orders. It estimates costs for both nested-loop joins and sort-merge joins.
- If no useful index exists on the inner table of a join, the optimizer may decide to build a temporary index, a process called **reformatting**.

  See “Reformatting strategy” on page 492.
- It determines the I/O size and caching strategy.
- It also compares the cost of serial and parallel execution, if parallel query processing is enabled.

  See Chapter 23, “Parallel Query Optimization,” for more information.

Factors that determine costs on single-table selects, such as appropriate indexing, search argument selectivity, and density of keys, become much more critical for joins.

## When statistics are not available for joins

If statistics are not available for a column in a join, the optimizer uses default values:

| Operator type | Examples | Default selectivity |
|---|---|---|
| Equality | `t1.c1 = t1.c2` | 1/rows in smaller table |
| Nonequality | `t1.c1 > t1.c2`<br>`t1.c1 >= t1.c2`<br>`t1.c1 < t1.c2`<br>`t1.c1 <= t1.c2` | 33% |

For example, in the following query, the optimizer uses 1/500 for the join selectivity for both tables if there are no statistics for either city column, and stores has 500 rows and authors has 5000 rows:

```
select au_fname, au_lname, stor_name
    from authors a, stores s
    where a.city = s.city
```

## Density values and joins

When statistics are available on a join column, the total density is used to estimate how many rows match each join key. If the authors table has 5000 rows, and the total density for the city column is .000879, the optimizer estimates that 5000 * .000879 = 4 rows will be returned from authors each time a join on the city column matches a row from the other table.

## Multiple column joins

When a join query specifies multiple join columns on two tables, and there is a composite index on the columns, the composite total density is used. For example, if authors and publishers each has an index on city, state, the composite total density for city, state is used for each table in this query:

```
select au_lname, pub_name
from authors a, publishers p
where a.city = p.city
and a.state = p.state
```

## Search arguments and joins on a table

When there are search arguments and joins on a table, the selectivities of the columns are combined during join costing to estimate the number of rows more accurately.

The following example joins authors and stores on both the city and state columns. There is a search argument on authors.state, so search argument transitive closure adds the search argument for stores.state table also:

```
select au_fname, au_lname, stor_name
from authors a, stores s
where  a.city = s.city
and a.state = s.state
and a.state = "GA"
```

If there is an index on city for each table, but no statistics available for state, the optimizer uses the default search argument selectivity (10%) combined with the total density for city. This overestimates the number of rows that match the search argument for this query, for a state with more rows that match a search argument on state, it would underestimate the number of rows. When statistics exist for state on each table, the estimate of the number of qualifying rows improves, and overall costing for the join query improves also.

# Datatype mismatches and query optimization

One common problem when queries fail to use indexes as expected is datatype mismatches. Datatype mismatches occur:

- With search clauses using variables or stored procedure parameters that have a different datatype than the column, for example:

  ```
  where int_col = @money_parameter
  ```

- In join queries when the columns being joined have different datatypes, for example:

  ```
  where tableA.int_col = tableB.money_col
  ```

Datatype mismatches lead to optimization problems when they prevent the optimizer from considering an index. The most common problems arise from:

- Comparisons between the integer types, int, smallint and tinyint
- Comparisons between money and smallmoney
- Comparisons between datetime and smalldatetime
- Comparisons between numeric and decimal types of differing precision and scale

- Comparisons between numeric or decimal types and integer or money columns

To avoid problems, use the same datatype (including the same precision and scale) for columns that are likely join candidates when you create tables. Use a matching datatype for any variables or stored procedure parameters used as search arguments. The following sections detail the rules and considerations applied when the same datatype is not used, and provide some troubleshooting tips.

## Overview of the datatype hierarchy and index issues

The datatype hierarchy controls the use of indexes when search arguments or join columns have different datatypes. The following query prints the hierarchy values and datatype names:

```
select hierarchy, name from systypes order by 1
hierarchy name
 --------- ------------------------------
         1 floatn
         2 float
         3 datetimn
         4 datetime
         5 real
         6 numericn
         7 numeric
         8 decimaln
         9 decimal
        10 moneyn
        11 money
        12 smallmoney
        13 smalldatetime
        14 intn
        15 int
        16 smallint
        17 tinyint
        18 bit
        19 univarchar
        20 unichar
        21 reserved
        22 varchar
        22 sysname
        22 nvarchar
        23 char
        23 nchar
```

```
24 varbinary
24 timestamp
25 binary
26 text
27 image
```

If you have created user-defined datatypes, they are also listed in the query output, with the corresponding hierarchy values.

The general rule is that when different datatypes are used, the systypes.hierarchy value determines whether an index can be used.

- For search arguments, the index is considered when the column's datatype is same as, or precedes, the hierarchy value of the parameter or variable.
- For a join, the index is considered only on the column whose systypes.hierarchy value is the same as the other column's, or precedes the other column's in the hierarchy.
- When char and unichar datatypes are used together, char is converted to unichar.

The exceptions are:

- Comparisons between char and varchar, unichar and univarchar, or between binary and varbinary datatypes. For example, although their hierarchy values are 23 and 22 respectively, char and varchar columns are treated as the same datatype for index consideration purposes. The index is considered for both columns in this join:

  ```
  where t1.char_column = t2.varchar_column
  ```

  char columns that accept NULL values are stored as varchar, but indexes can still be used on both columns for joins.
- The null type of the column has no effect, that is, although float and floatn have different hierarchy values, they are always treated as the same datatype.
- Comparisons of decimal or numeric types also take precision and scale into account. This includes comparisons of numeric or decimal types to each other, and comparisons of numeric or decimal to other datatypes such as int or money.

  See "Comparison of numeric and decimal datatypes" on page 398 for more information.

## Comparison of numeric and decimal datatypes

When a query joins columns of numeric or decimal datatypes, an index can be used when both of these conditions are true:

- The scale of the column being considered for a join equals or exceeds the scale of the other join column, and
- The length of the integer portion of the column equals or exceeds the length of the other column's integer portion.

Here are some examples of when indexes can be considered:

| Datatypes in the join | Indexes considered |
|---|---|
| numeric(12,4) and numeric(16,4) | Index considered only for numeric(16,4), the integer portion of numeric(12,4) is smaller. |
| numeric(12,4) and numeric(12,8) | Neither index is considered, integer portion is smaller for numeric(12,8) and scale is smaller for numeric(12,4). |
| numeric(12,4) and numeric(12,4) | Both indexes are considered. |

## Comparing numeric types to other datatypes

When comparing numeric and decimal columns to columns of other numeric datatypes, such as money or int:

- numeric and decimal precede integer and money columns in the hierarchy, so the index on the numeric or decimal column is the only index considered.
- The precision and scale requirements must be met for the numeric or decimal index to be considered. The scale of the numeric column must be equal to, or greater than, the scale of the integer or money column, and the number of digits in the integer portion of the numeric column must be equal to or greater than the maximum number of digits usable for the integer or money column.

The precision and scale of integer and money types is shown in Table 17-3.

***Table 17-3: Precision and scale of integer and money types***

| Datatype | Precision, scale |
|---|---|
| tinyint | 3,0 |
| smallint | 5,0 |
| int | 10,0 |
| smallmoney | 10,4 |
| money | 19,4 |

## Datatypes for parameters and variables used as SARGs

When declaring datatypes for variables or stored procedure parameters to be used as search arguments, match the datatype of the column in the variable or parameter declaration to ensure the use of an index. For example:

```
declare @int_var int
select @int_var = 50
select *
from t1
where int_col = @int_var
```

Use of the index depends on the precedence of datatypes in the hierarchy. The index on a column can be used only if the column's datatype precedes the variable's datatype. For example, int precedes smallint and tinyint in the hierarchy. Here are just the integer types:

```
hierarchy  name
 --------- ------------------------------
        15 int
        16 smallint
        17 tinyint
```

If a variable or parameter has a datatype of smallint or tinyint, an index on an int column can be used for a query. But an index on a tinyint column cannot be used for an int parameter.

Similarly, money precedes int. If a variable or parameter of money is compared to an int column, an index on the int column cannot be used.

This eliminates issues that could arise from truncation or overflow. For example, it would not be useful or correct to attempt to truncate the money value to 5 in order to use an index on *int_col* for this query:

```
declare @money_var money
select @money_var = $5.12
select * from t1 where int_col = @money_var
```

### Troubleshooting datatype mismatch problems fo SARGs

If there is a datatype mismatch problem with a search argument on an indexed column, the query can use another index if there are other search arguments or it can perform a table scan. showplan output displays the access method and keys used for each table in a query.

You can use dbcc traceon(302) to determine whether an index is being considered. For example, using an integer variable as a search argument on int_col produces the following output:

```
Selecting best index for the SEARCH CLAUSE:
    t1.int_col = unknown-value

SARG is a local variable or the result of a function or
an expression, using the total density to estimate
selectivity.

Estimated selectivity for int_col,
   selectivity = 0.020000.
```

Using an incompatible datatype such as money for a variable used as a search argument on an integer column does not produce a "Selecting best index for the SEARCH CLAUSE" block in dbcc traceon(302) output, indicating that the index is not being considered, and cannot be used. If an index is not used as you expect in a query, looking for this costing section in dbcc traceon(302) output should be one of your first debugging steps.

The "unknown-value" and the fact that the total density is used to estimate the number of rows that match this search argument is due to the fact that the value of the variable was set in the batch; it is not a datatype mismatch problem.

See "SARGs using variables and parameters" on page 392 for more information.

## Compatible datatypes for join columns

The optimizer considers an index for joined columns only when the column types are the same or when the datatype of the join column precedes the other column's datatype in the datatype hierarchy. This means that the optimizer considers using the index on only one of the join columns, limiting the choice of join orders.

For example, this query joins columns of decimal and int datatypes:

```
select *
```

```
from t1, t2
where t1.decimal_col = t2.int_col
```

decimal precedes *int* in the hierarchy, so the optimizer can consider an index on t1.decimal_col, but cannot use an index on t2.int_col. The result is likely to be a table scan of t2, followed by use of the index on t1.decimal_col.

Table 17-4 shows how the hierarchy affects index choice for some commonly problematic datatypes.

***Table 17-4: Indexes considered for mismatched column datatypes***

| Join column types | Index considered on column of type |
|---|---|
| money and smallmoney | money |
| datetime and smalldatetime | datetime |
| int and smallint | int |
| int and tinyint | int |
| smallint and tinyint | smallint |

### Troubleshooting datatype mismatch problems for joins

If you suspect that an index is not being considered on one side of a join due to datatype mismatches, use dbcc traceon(302). In the output, look for "Selecting best index for the JOIN CLAUSE". If datatypes are compatible, you see two of these blocks for each join; for example:

```
Selecting best index for the JOIN CLAUSE:
    t1.int_col = t2.int_col
```

And later in the output for the other table in the join:

```
Selecting best index for the JOIN CLAUSE:
    t2.int_col = t1.int_col
```

For a query that compares incompatible datatypes, for example, comparing a decimal column to an int, column, there is only the single block:

```
Selecting best index for the JOIN CLAUSE:
    t1.decimal_col = t2.int_col
```

This means that the join costing for using an index with t2.int_col as the outer column is not performed.

## Suggestions on datatypes and comparisons

To avoid datatype mismatch problems:

- When you create tables, use the same datatypes for columns that will be joined.
- If columns of two frequently joined tables have different datatypes, consider using alter table...modify to change the datatype of one of the columns.
- Use the column's datatype whenever declaring variables or stored procedure parameters that will be used as search arguments.
- Consider user-defined datatype definitions. Once you have created definitions with sp_addtype, you can use them in commands such create table, alter table, and create procedure, and for datatype declarations.
- For some queries where datatype mismatches cause performance problems, you may be able to use the convert function so that indexes are considered on the other table in the join. The next section describes this work around.

## Forcing a conversion to the other side of a join

If a join between different datatypes is unavoidable, and it impacts performance, you can, for some queries, force the conversion to the other side of the join. In the following query, an index on smallmoney_col cannot be used, so the query performs a table scan on huge_table:

```
select *
from tiny_table, huge_table
where tiny_table.money_col =
    huge_table.smallmoney_col
```

Performance improves if the index on huge_table.smallmoney_col can be used. Using the convert function on the money column of the small table allows the index on the large table to be used, and a table scan is performed on the small table:

```
select *
from tiny_table, huge_table
where convert(smallmoney,tiny_table.money_col) =
    huge_table.smallmoney_col
```

This workaround assumes that there are no values in tinytable.money_col that are large enough to cause datatype conversion errors during the conversion to smallmoney. If there are values larger than the maximum value for smallmoney, you can salvage this solution by adding a search argument specifying the maximum values for a smallmoney column:

```
select smallmoney_col, money_col
from tiny_table , huge_table
where convert(smallmoney,tiny_table.money_col) =
    huge_table.smallmoney_col
and tiny_table.money_col <= 214748.3647
```

Converting floating-point and numeric data can change the meaning of some queries. This query compares integers and floating-point numbers:

```
select *
    from tab1, tab2
    where tab1.int_column = tab2.float_column
```

In the query above,you cannot use an index on int_column. This conversion forces the index access to tab1, but also returns different results than the query that does not use convert:

```
select *
from tab1, tab2
where tab1.int_col = convert(int, tab2.float_col)
```

For example, if int_column is 4, and float_column is 4.2, the modified query implicitly converts to a 4, and returns a row not returned by the original query. The workaround can be salvaged by adding this self-join:

```
and tab2.float_col = convert(int, tab2.float_col)
```

This workaround assumes that all values in tab2.float_col can be converted to int without conversion errors.

# Splitting stored procedures to improve costing

The optimizer cannot use statistics the final select in the following procedure, because it cannot know the value of *@city* until execution time:

```
create procedure au_city_names
    @pub_name varchar(30)
as
    declare @city varchar(25)
    select @city = city
    from publishers where pub_name = @pub_name
    select au_lname
        from authors
        where city = @city
```

The following example shows the procedure split into two procedures. The first procedure calls the second one:

```
create procedure au_names_proc
    @pub_name varchar(30)
as
    declare @city varchar(25)
    select @city = city
        from publishers
        where pub_name = @pub_name
    exec select_proc @city
create procedure select_proc @city varchar(25)
as
    select au_lname
        from authors
        where city = @city
```

When the second procedure executes, Adaptive Server knows the value of *@city* and can optimize the select statement. Of course, if you modify the value of *@city* in the second procedure before it is used in the select statement, the optimizer may choose the wrong plan because it optimizes the query based on the value of *@city* at the start of the procedure. If *@city* has different values each time the second procedure is executed, leading to very different query plans, you may want to use with recompile.

# Basic units of costing

When the optimizer estimates costs for the query, the two factors it considers are the cost of physical I/O, reading pages from disk, and the cost of logical I/O, finding pages in the data cache. The optimizer assigns 18 as the cost of a physical I/O and 2 as the cost of a logical I/O. These are relative units of cost and do not represent time units such as milliseconds or clock ticks. These units are used in the formulas in this chapter, with the physical I/O costs first, then the logical I/O costs. The total cost of accessing a table can be expressed as:

Cost = All physical IOs * 18 + All logical IOs * 2

CHAPTER 18

# Advanced Optimizing Tools

This chapter describes query processing options that affect the optimizer's choice of join order, index, I/O size and cache strategy.



## Special optimizing techniques

Being familiar with the information presented in the *Basics* volume helps to understand the material in this chapter. Use caution, as the tools allow you to override the decisions made by Adaptive Server's optimizer and can have an extreme negative effect on performance if misused. You should understand the impact on the performance of both your individual query and the possible implications for overall system performance.

Adaptive Server's advanced, cost-based optimizer produces excellent query plans in most situations. But there are times when the optimizer does not choose the proper index for optimal performance or chooses a suboptimal join order, and you need to control the access methods for the query. The options described in this chapter allow you that control.

In addition, while you are tuning, you may want to see the effects of a different join order, I/O size, or cache strategy. Some of these options let you specify query processing or access strategy without costly reconfiguration.

Adaptive Server provides tools and query clauses that affect query optimization and advanced query analysis tools that let you understand why the optimizer makes the choices that it does.

**Note** This chapter suggests workarounds for certain optimization problems. If you experience these types of problems, please call Sybase Technical Support.

# Specifying optimizer choices

Adaptive Server lets you specify these optimization choices by including commands in a query batch or in the text of the query:

- The order of tables in a join
- The number of tables evaluated at one time during join optimization
- The index used for a table access
- The I/O size
- The cache strategy
- The degree of parallelism

In a few cases, the optimizer fails to choose the best plan. In some of these cases, the plan it chooses is only slightly more expensive than the “best” plan, so you need to weigh the cost of maintaining forced options against the slower performance of a less than optimal plan.

The commands to specify join order, index, I/O size, or cache strategy, coupled with the query-reporting commands like statistics io and showplan, can help you determine why the optimizer makes its choices.

**Warning!** Use the options described in this chapter with caution. The forced query plans may be inappropriate in some situations and may cause very poor performance. If you include these options in your applications, check query plans, I/O statistics, and other performance data regularly.

These options are generally intended for use as tools for tuning and experimentation, not as long-term solutions to optimization problems.

# Specifying table order in joins

Adaptive Server optimizes join orders to minimize I/O. In most cases, the order that the optimizer chooses does not match the order of the from clauses in your select command. To force Adaptive Server to access tables in the order they are listed, use:

```
set forceplan [on|off]
```

The optimizer still chooses the best access method for each table. If you use forceplan, specifying a join order, the optimizer may use different indexes on tables than it would with a different table order, or it may not be able to use existing indexes.

You might use this command as a debugging aid if other query analysis tools lead you to suspect that the optimizer is not choosing the best join order. Always verify that the order you are forcing reduces I/O and logical reads by using set statistics io on and comparing I/O with and without forceplan.

If you use forceplan, your routine performance maintenance checks should include verifying that the queries and procedures that use it still require the option to improve performance.

You can include forceplan in the text of stored procedures.

set forceplan forces only join order, and not join type. There is no command for specifying the join type; you can disable merge joins at the server or session level.

See “Enabling and disabling merge joins” on page 417 for more information.

## Risks of using *forceplan*

Forcing join order has these risks:

- Misuse can lead to extremely expensive queries. Always test the query thoroughly with statistics io, and with and without forceplan.
- It requires maintenance. You must regularly check queries and stored procedures that include forceplan. Also, future versions of Adaptive Server may eliminate the problems that lead you to incorporate index forcing, so you should check all queries using forced query plans each time a new version is installed.

## Things to try before using *forceplan*

Before you use forceplan:

- Check showplan output to determine whether index keys are used as expected.
- Use dbcc traceon(302) to look for other optimization problems.
- Run update statistics on the index.
- Use update statistics to add statistics for search arguments on unindexed search clauses in the query, especially for search arguments that match minor keys in compound indexes.
- If the query joins more than four tables, use set table count to see if it results in an improved join order.

  See “Specifying the number of tables considered by the optimizer” on page 409.

# Specifying the number of tables considered by the optimizer

Adaptive Server optimizes joins by considering permutations of two to four tables at a time, as described in "Costing and optimizing joins" on page 471. If you suspect that an inefficient join order is being chosen for a join query, you can use the set table count option to increase the number of tables that are considered at the same time. The syntax is:

```
set table count int_value
```

Valid values are 0 though 8; 0 restores the default behavior.

For example, to specify 4-at-a-time optimization, use:

```
set table count 4
```

dbcc traceon(310) reports the number of tables considered at a time. See "dbcc traceon(310) and final query plan costs" on page 891 for more information.

As you decrease the value, you reduce the chance that the optimizer will consider all the possible join orders. Increasing the number of tables considered at one time during join ordering can greatly increase the time it takes to optimize a query.

Since the time it takes to optimize the query is increased with each additional table, the set table count option is most useful when the execution savings from improved join order outweighs the extra optimizing time. Some examples are:

- If you think that a more optimal join order can shorten total query optimization and execution time, especially for stored procedures that you expect to be executed many times once a plan is in the procedure cache
- When saving abstract plans for later use

Use statistics time to check parse and compile time and statistics io to verify that the improved join order is reducing physical and logical I/O.

If increasing the table count produces an improvement in join optimization, but increases the CPU time unacceptably, rewrite the from clause in the query, specifying the tables in the join order indicated by showplan output, and use forceplan to run the query. Your routine performance maintenance checks should include verifying that the join order you are forcing still improves performance.

# Specifying an index for a query

You can specify the index to use for a query using the (index *index_name*) clause in select, update, and delete statements. You can also force a query to perform a table scan by specifying the table name. The syntax is:

```
select select_list
   from table_name [correlation_name]
      (index {index_name | table_name } )
      [, table_name ...]
   where ...


delete table_name
   from table_name [correlation_name]
   (index {index_name | table_name }) ...


update table_name set col_name = value
   from table_name [correlation_name]
   (index {index_name | table_name})...
```

For example:

```
select pub_name, title
    from publishers p, titles t (index date_type)
    where p.pub_id = t.pub_id
    and type = "business"
    and pubdate > "1/1/93"
```

Specifying an index in a query can be helpful when you suspect that the optimizer is choosing a suboptimal query plan. When you use this option:

- Always check statistics io for the query to see whether the index you choose requires less I/O than the optimizer's choice.
- Ttest a full range of valid values for the query clauses, especially if you are tuning queries:
  - Tuning queries on tables that have skewed data distribution
  - Performing range queries, since the access methods for these queries are sensitive to the size of the range

Use this option only after testing to be certain that the query performs better with the specified index option. Once you include an index specification in a query, you should check regularly to be sure that the resulting plan is still better than other choices made by the optimizer.

---

**Note** If a nonclustered index has the same name as the table, specifying a table name causes the nonclustered index to be used. You can force a table scan using select *select_list* from *tablename* (0).

---

## Risks

Specifying indexes has these risks:

- Changes in the distribution of data could make the forced index less efficient than other choices.
- Dropping the index means that all queries and procedures that specify the index print an informational message indicating that the index does not exist. The query is optimized using the best alternative access method.
- Maintenance increases, since all queries using this option need to be checked periodically. Also, future versions of Adaptive Server may eliminate the problems that lead you to incorporate index forcing, so you should check all queries using forced indexes each time you install a new version.

## Things to try before specifying an index

Before specifying an index in queries:

- Check showplan output for the "Keys are" message to be sure that the index keys are being used as expected.
- Use dbcc traceon(302) to look for other optimization problems.
- Run update statistics on the index.

- If the index is a composite index, run update statistics on the minor keys in the index, if they are used as search arguments. This can greatly improve optimizer cost estimates. Creating statistics for other columns frequently used for search clauses can also improve estimates.

# Specifying I/O size in a query

If your Adaptive Server is configured for large I/Os in the default data cache or in named data caches, the optimizer can decide to use large I/O for:

- Queries that scan entire tables
- Range queries using clustered indexes, such as queries using >, <, > *x* and < *y*, between, and like "*charstring %*"
- Queries that scan a large number of index leaf pages

If the cache used by the table or index is configured for 16K I/O, a single I/O can read up to eight pages simultaneously. Each named data cache can have several pools, each with a different I/O size. Specifying the I/O size in a query causes the I/O for that query to take place in the pool that is configured for that size. See the *System Administration Guide* for information on configuring named data caches.

To specify an I/O size that is different from the one chosen by the optimizer, add the prefetch specification to the index clause of a select, delete, or update statement. The syntax is:

```
select select_list
    from table_name
      ( [index {index_name | table_name} ]
          prefetch size)
      [, table_name ...]
where ...


delete table_name from table_name
    ( [index {index_name | table_name} ]
        prefetch size)
...
```

```
update table_name set col_name = value
   from table_name
      ( [index {index_name | table_name} ]
         prefetch size)
...
```

Valid values for *size* are 2, 4, 8, and 16. If no pool of the specified size exists in the data cache used by the object, the optimizer chooses the best available size.

If there is a clustered index on au_lname, this query performs 16K I/O while it scans the data pages:

```
select *
from authors (index au_names prefetch 16)
    where au_lname like "Sm%"
```

If a query normally performs large I/O, and you want to check its I/O performance with 2K I/O, you can specify a size of 2K:

```
select type, avg(price)
    from titles (index type_price prefetch 2)
    group by type
```

## Index type and large I/O

When you specify an I/O size with prefetch, the specification can affect both the data pages and the leaf-level index pages. Table 18-1 shows the effects.

***Table 18-1: Access methods and prefetching***

| Access method | Large I/O performed on |
|---|---|
| Table scan | Data pages |
| Clustered index | Data pages only, for allpages-locked tables |
| | Data pages and leaf-level index pages for data-only-locked tables |
| Nonclustered index | Data pages and leaf pages of nonclustered index |

showplan reports the I/O size used for both data and leaf-level pages.

See “I/O Size Messages” on page 812 for more information.

## When *prefetch* specification is not followed

In most cases, when you specify an I/O size in a query, the optimizer incorporates the I/O size into the query's plan. However, there are times when the specification cannot be followed, either for the query as a whole or for a single, large I/O request.

Large I/O cannot be used for the query if:

- The cache is not configured for I/O of the specified size. The optimizer substitutes the best size available.
- sp_cachestrategy has been used to disable large I/O for the table or index.

Large I/O cannot be used for a single buffer if

- Any of the pages included in that I/O request are in another pool in the cache.
- The page is on the first extent in an allocation unit. This extent holds the allocation page for the allocation unit, and only seven data pages.
- No buffers are available in the pool for the requested I/O size.

Whenever a large I/O cannot be performed, Adaptive Server performs 2K I/O on the specific page or pages in the extent that are needed by the query.

To determine whether the prefetch specification is followed, use showplan to display the query plan and statistics io to see the results on I/O for the query. sp_sysmon reports on the large I/Os requested and denied for each cache.

See "Data cache management" on page 973.

## *set prefetch on*

By default, a query uses large I/O whenever a large I/O pool is configured and the optimizer determines that large I/O would reduce the query cost. To disable large I/O during a session, use:

```
set prefetch off
```

To reenable large I/O, use:

```
set prefetch on
```

If large I/O is turned off for an object using sp_cachestrategy, set prefetch on does not override that setting.

If large I/O is turned off for a session using set prefetch off, you cannot override the setting by specifying a prefetch size as part of a select, delete, or insert statement.

The set prefetch command takes effect in the same batch in which it is run, so you can include it in a stored procedure to affect the execution of the queries in the procedure.

## Specifying the cache strategy

For queries that scan a table's data pages or the leaf level of a nonclustered index (covered queries), the Adaptive Server optimizer chooses one of two cache replacement strategies: the fetch-and-discard (MRU) strategy or the LRU strategy.

See "Overview of cache strategies" on page 162 for more information about these strategies.

The optimizer may choose the fetch-and-discard (MRU) strategy for:

- Any query that performs table scans
- A range query that uses a clustered index
- A covered query that scans the leaf level of a nonclustered index
- An inner table in a nested-loop join, if the inner table is larger than the cache
- The outer table of a nested-loop join, since it needs to be read only once
- Both tables in a merge join

You can affect the cache strategy for objects:

- By specifying lru or mru in a select, update, or delete statement
- By using sp_cachestrategy to disable or reenable mru strategy

If you specify MRU strategy, and a page is already in the data cache, the page is placed at the MRU end of the cache, rather than at the wash marker.

Specifying the cache strategy affects only data pages and the leaf pages of indexes. Root and intermediate pages always use the LRU strategy.

### In *select*, *delete*, and *update* statements

You can use lru or mru (fetch-and-discard) in a select, delete, or update command to specify the I/O size for the query:

```
select select_list
    from table_name
        (index index_name prefetch size [lru|mru])
        [, table_name ...]
where ...


delete table_name from table_name (index index_name
    prefetch size [lru|mru]) ...


update table_name set col_name = value
    from table_name (index index_name
        prefetch size [lru|mru]) ...
```

This query adds the LRU replacement strategy to the 16K I/O specification:

```
select au_lname, au_fname, phone
    from authors (index au_names prefetch 16 lru)
```

For more information about specifying a prefetch size, see “Specifying I/O size in a query” on page 412.

# Controlling large I/O and cache strategies

Status bits in the sysindexes table identify whether a table or an index should be considered for large I/O prefetch or for MRU replacement strategy. By default, both are enabled. To disable or reenable these strategies, use sp_cachestrategy. The syntax is:

```
sp_cachestrategy dbname , [ownername.]tablename
    [, indexname | "text only" | "table only"
    [, { prefetch | mru }, { "on" | "off"}]]
```

This command turns off the large I/O prefetch strategy for the au_name_index of the authors table:

```
sp_cachestrategy pubtune,
authors, au_name_index, prefetch, "off"
```

This command reenables MRU replacement strategy for the titles table:

```
                    sp_cachestrategy pubtune,
titles, "table only", mru, "on"
```

Only a System Administrator or the object owner can change or view the cache strategy status of an object.

### Getting information on cache strategies

To see the cache strategy that is in effect for a given object, execute sp_cachestrategy, with the database and object name:

```
sp_cachestrategy pubtune, titles
object name      index name       large IO MRU
---------------- ---------------- -------- --------
titles           NULL             ON       ON
```

showplan output shows the cache strategy used for each object, including worktables.

## Enabling and disabling merge joins

By default, merge joins are not enabled at the server level. When merge joins are disabled, the server only costs nested-loop joins, and merge joins are not considered. To enable merge joins server-wide, set enable sort-merge joins and JTC to 1. This also enables join transitive closure.

The command set sort_merge on overrides the server level to allow use of merge joins in a session or stored procedure.

To enable merge joins, use:

```
set sort_merge on
```

To disable merge joins, use:

```
set sort_merge off
```

For information on configuring merge joins server-wide see the *System Administration Guide*.

# Enabling and disabling join transitive closure

By default, join transitive closure is not enabled at the server level, since it can increase optimization time. You can enable join transitive closure at a session level with set jtc on. The session-level command overrides the server-level setting for the enable sort-merge joins and JTC configuration parameter.

For queries that execute quickly, even when several tables are involved, join transitive closure may increase optimization time with little improvement in execution cost. For example, with join transitive closure applied to this query, the number of possible joins is multiplied for each added table:

```
select * from t1, t2, t3, t4, ... tN
where t1.c1 = t2.c1
and t1.c1 = t3.c1
and t1.c1 = t4.c1
...
and t1.c1 = tN.c1
```

For joins on very large tables, however, the additional optimization time involved in costing the join orders added by join transitive closure may result in a join order that greatly improves the response time.

You can use set statistics time to see how long it takes to optimize the query. If running queries with set jtc on greatly increases optimization time, but also improves query execution by choosing a better join order, check the showplan or dbcc traceon(302, 310) output. Explicitly add the useful join orders to the query text. You can run the query without join transitive closure, and get the improved execution time, without the increased optimization time of examining all possible join orders generated by join transitive closure.

You can also enable join transitive closure and save abstract plans for queries that benefit. If you then execute those queries with loading from the saved plans enabled, the saved execution plan is used to optimize the query, making optimization time extremely short.

See Chapter 28, “Introduction to Abstract Plans,” for more information on using abstract plans.

For information on configuring join transitive closure server-wide see the *System Administration Guide*.

# Suggesting a degree of parallelism for a query

The parallel and *degree_of_parallelism* extensions to the from clause of a select command allow users to restrict the number of worker processes used in a scan.

For a parallel partition scan to be performed, the *degree_of_parallelism* must be equal to or greater than the number of partitions. For a parallel index scan, specify any value for the *degree_of_parallelism*.

The syntax for the select statement is:

```
select...
    [from {tablename}
        [(index index_name
            [parallel [degree_of_parallelism | 1]]
            [prefetch size] [lru|mru])],
        {tablename} [([index_name]
            [parallel [degree_of_parallelism | 1]
              [prefetch size] [lru|mru])] ...
```

Table 18-2 shows how to combine the index and parallel keywords to obtain serial or parallel scans.

***Table 18-2: Optimizer hints for serial and parallel execution***

| To specify this type of scan: | Use this syntax: |
|---|---|
| Parallel partition scan | (index *tablename* parallel *N*) |
| Parallel index scan | (index *index_name* parallel *N*) |
| Serial table scan | (index *tablename* parallel 1) |
| Serial index scan | (index *index_name* parallel 1) |
| Parallel, with the choice of table or index scan left to the optimizer | (parallel *N*) |
| Serial, with the choice of table or index scan left to the optimizer | (parallel 1) |

When you specify the parallel degree for a table in a merge join, it affects the degree of parallelism used for both the scan of the table and the merge join.

You cannot use the parallel option if you have disabled parallel processing either at the session level with the set parallel_degree 1 command or at the server level with the parallel degree configuration parameter. The parallel option cannot override these settings.

If you specify a *degree_of_parallelism* that is greater than the maximum configured degree of parallelism, Adaptive Server ignores the hint.

The optimizer ignores hints that specify a parallel degree if any of the following conditions is true:

- The from clause is used in the definition of a cursor.
- parallel is used in the from clause of an inner query block of a subquery, and the optimizer does not move the table to the outermost query block during subquery flattening.
- The table is a view, a system table, or a virtual table.
- The table is the inner table of an outer join.
- The query specifies exists, min, or max on the table.
- The value for the max scan parallel degree configuration parameter is set to 1.
- An unpartitioned clustered index is specified or is the only parallel option.
- A nonclustered index is covered.
- The query is processed using the OR strategy.

  For an explanation of the OR strategy, see "Access Methods and Costing for or and in Clauses" on page 451.
- The select statement is used for an update or insert.

## Query level *parallel* clause examples

To specify the degree of parallelism for a single query, include parallel after the table name. This example executes in serial:

```
select * from titles (parallel 1)
```

This example specifies the index to be used in the query, and sets the degree of parallelism to 5:

```
select * from titles
    (index title_id_clix parallel 5)
where ...
```

To force a table scan, use the table name instead of the index name.

# Concurrency optimization for small tables

For data-only-locked tables of 15 pages or fewer, Adaptive Server does not consider a table scan if there is a useful index on the table. Instead, it always chooses the cheapest index that matches any search argument that can be optimized in the query. The locking required for an index scan provides higher concurrency and reduces the chance of deadlocks, although slightly more I/O may be required than for a table scan.

If concurrency on small tables is not an issue, and you want to optimize the I/O instead, you can disable this optimization with sp_chgattribute. This command turns off concurrency optimization for a table:

```
sp_chgattribute tiny_lookup_table,
    "concurrency_opt_threshold", 0
```

With concurrency optimization disabled, the optimizer can choose table scans when they require fewer I/Os.

You can also increase the concurrency optimization threshold for a table. This command sets the concurrency optimization threshold for a table to 30 pages:

```
sp_chgattribute lookup_table,
    "concurrency_opt_threshold", 30
```

The maximum value for the concurrency optimization threshold is 32,767. Setting the value to -1 enforces concurrency optimization for a table of any size. It may be useful in cases where a table scan is chosen over indexed access, and the resulting locking results in increased contention or deadlocks.

The current setting is stored in systabstats.conopt_thld and is printed as part of optdiag output.

## Changing locking scheme

Concurrency optimization affects only data-only-locked tables. Table 18-3 shows the effect of changing the locking scheme.

***Table 18-3: Effects of alter table on concurrency optimization settings***

| Changing locking scheme from | Effect on stored value |
|---|---|
| Allpages to data-only | Set to 15, the default |
| Data-only to allpages | Set to 0 |

| Changing locking scheme from | Effect on stored value |
|---|---|
| One data-only scheme to another | Configured value retained |

CHAPTER 19

# Query Tuning Tools

This chapter provides a guide to the tools that can help you tune your queries.



The tools mentioned in this chapter are described in more detail in the chapters that follow.

## Overview

Adaptive Server provides the following diagnostic and informational tools to help you understand query optimization and improve the performance of your queries:

- A choice of tools to check or estimate the size of tables and indexes. These tools are described in Chapter 15, “Determining Sizes of Tables and Indexes.”
- set statistics io on displays the number of logical and physical reads and writes required for each table in a query. If resource limits are enabled, it also displays the total actual I/O cost. set statistics io is described in Chapter 34, “Using the set statistics Commands.”
- set showplan on displays the steps performed for each query in a batch. It is often used with set noexec on, especially for queries that return large numbers of rows.

  See Chapter 35, “Using set showplan.”
- set statistics subquerycache on displays the number of cache hits and misses and the number of rows in the cache for each subquery.

  See “Subquery results caching” on page 502 for examples.

- set statistics time on displays the time it takes to parse and compile each command.

  See “Checking compile and execute time” on page 762 for more information.

- dbcc traceon (302) and dbcc traceon(310) provide additional information about why particular plans were chosen and is often used when the optimizer chooses a plan that seems incorrect.

  See Chapter 37, “Tuning with dbcc traceon.”

- The optdiag utility command displays statistics for tables, indexes, and columns.

  See Chapter 36, “Statistics Tables and Displaying Statistics with optdiag.”

- Chapter 18, “Advanced Optimizing Tools,” explains tools you can use to enforce index choice, join order, and other query optimization choices. These tools include:

  - set forceplan – forces the query to use the tables in the order specified in the from clause.
  - set table count – increases the number of tables that the optimizer considers at one time while determining join order.
  - select, delete, update clauses with (index...prefetch...mru_lru...parallel) –specifies the index, I/O size, or cache strategy to use for the query.
  - set prefetch –toggles prefetch for query tuning experimentation.
  - set sort_merge – disallows sort-merge joins.
  - set parallel_degree – specifies the degree of parallelism for a query.
  - sp_cachestrategy – sets status bits to enable or disable prefetch and fetch-and-discard cache strategies.

# How tools may interact

showplan, statistics io, and other commands produce their output while stored procedures are being run. The system procedures that you might use for checking table structure or indexes as you test optimization strategies can produce voluminous output when diagnostic information is being printed. You may want to have hard copies of your table schemas and index information, or you can use separate windows for running system procedures such as sp_helpindex.

For lengthy queries and batches, you may want the save showplan and statistics io output in files. You can do so by using "echo input" flag to isql. The syntax is:

isql -P *password* -e -i *input_file* -o *outputfile*

## Using *showplan* and *noexec* together

showplan is often used in conjunction with set noexec on, which prevents SQL statements from being executed. Issue showplan, or any other set commands, before you issue the noexec command. Once you issue set noexec on, the only command that Adaptive Server executes is set noexec off. This example shows the correct order:

```
set showplan on
set noexec on
go
select au_lname, au_fname
    from   authors
    where  au_id = "A137406537"
go
```

## *noexec* and *statistics io*

While showplan and noexec make useful companions, noexec stops all the output of statistics io. The statistics io command reports actual disk I/O; while noexec is in effect, no I/O takes place, so the reports are not printed.

# How tools relate to query processing

Many of the tools, for example, the set commands, affect the decisions made by the optimizer. showplan and dbcc traceon(302, 310) show you optimizer decision-making. dbcc traceon(302,310) shows intermediate information as analysis is performed, with dbcc traceon(310) printing the final plan statistics. showplan shows the final decision on access methods and join order.

statistics io and statistics time provide information about how the query was executed: statistics time measures time from the parse step until the query completes. statistics io prints actual I/O performed during query execution.

noexec allows you to obtain information such as showplan or dbcc traceon(302,310) output without actually executing the query.

CHAPTER 20

# Access Methods and Query Costing for Single Tables

This chapter introduces the methods that Adaptive Server uses to access rows in tables. It examines various types of queries on single tables, and describes the access methods that can be used, and the associated costs.



Chapter 17, "Adaptive Server Optimizer," explains how the optimizer uses search arguments and join clauses to estimate the number of rows that a query will return. This chapter looks at how the optimizer uses row estimates and other statistics to estimate the number of pages that must be read for the query, and how many logical and physical I/Os are required.

This chapter looks at queries that affect a single table.

For queries that involve more than one table, see Chapter 21, "Accessing Methods and Costing for Joins and Subqueries."

For parallel queries, see Chapter 23, "Parallel Query Optimization."

This chapter contains information about query processing that you can use in several ways as it:

- Provides a general overview of the access methods that Adaptive Server uses to process a variety of queries, including illustrations and sample queries. This information will help you understand how particular types of queries are executed and how you can improve query performance by adding indexes or statistics for columns used in the queries.

- Provides a description of how the optimizer arrives at the logical and physical I/O estimates for the queries. These descriptions can help you understand whether the I/O use and response time are reasonable for a given query. These descriptions can be used with the following tuning tools:
  - optdiag can be used to display the statistics about your tables, indexes, and column values.

    See Chapter 36, "Statistics Tables and Displaying Statistics with optdiag."
  - showplan displays the access method (table scan, index scan, type of OR strategy, and so forth) for a query.

    See Chapter 35, "Using set showplan."
  - statistics io displays the logical and physical I/O for each table in a query.
- Provides detailed formulas, very close to the actual formulas used by Adaptive Server. Use these formulas are meant to be used in conjunction with the tuning tools:
  - optdiag can be used to display the statistics that you need to apply the formulas. See Chapter 36, "Statistics Tables and Displaying Statistics with optdiag."
  - dbcc traceon(302) displays the sizes, densities, selectivities and cluster ratios used to produce logical I/O estimates, and dbcc traceon(310) displays the final query costing for each table, including the estimated physical I/O. See Chapter 37, "Tuning with dbcc traceon."

  In many cases, you will need to use these formulas only when you are debugging problem queries. You may need to discover why an or query performs a table scan, or why an index that you thought was useful is not being used by a query.

This chapter can also help you determine when to stop working to improve the performance of a particular query. If you know that it needs to read a certain number of index pages and data pages, and the number of I/Os cannot be reduced further by adding a covering index, you know that you have reached the optimum performance possible for query analysis and index selection. You might need to look at other issues, such as cache configuration, parallel query options, or object placement.

# Table scan cost

When a query requires a table scan, Adaptive Server reads each page of the table from disk into the data cache and checks the data values (if there is a where clause) and returns qualifying rows.

Table scans are performed:

- When no index exists on the columns used in the search clauses.
- When the optimizer determines that using the index is more expensive than performing a table scan. The optimizer may determine that it is cheaper to read the data pages directly than to read the index pages and then the data pages for each row that is to be returned.

The cost of a table scan depends on the size of the table and the I/O size.

## Cost of a scan on allpages-locked table

The I/O cost of a table scan on an allpages-locked table using 2K I/O is one physical I/O and one logical I/O for each page in the table:

```
Table scan cost =       Number of pages * 18
                  +     Number of pages * 2
```

If the table uses a cache with large I/O, the number of physical I/Os is estimated by dividing the number of pages by the I/O size and using a factor that is based on the data page cluster ratio to estimate the number of large I/Os that need to be performed. Since large I/O cannot be performed on any data pages on the first extent in the allocation unit, each of those pages must be read with 2K I/O.

The logical I/O cost is one logical I/O for each page in the table. The formula is:

```
Table scan cost =       (pages /pages per IO) * Clustering adjust-
ment* 18+ Number of pages * 2
```

See “How cluster ratios affect large I/O estimates” on page 433 for more information on cluster ratios.

**Note** Adaptive Server does not track the number of pages in the first extent of an allocation unit for an allpages-locked table, so the optimizer does not include this slight additional I/O in its estimates.

## Cost of a scan on a data-only-locked tables

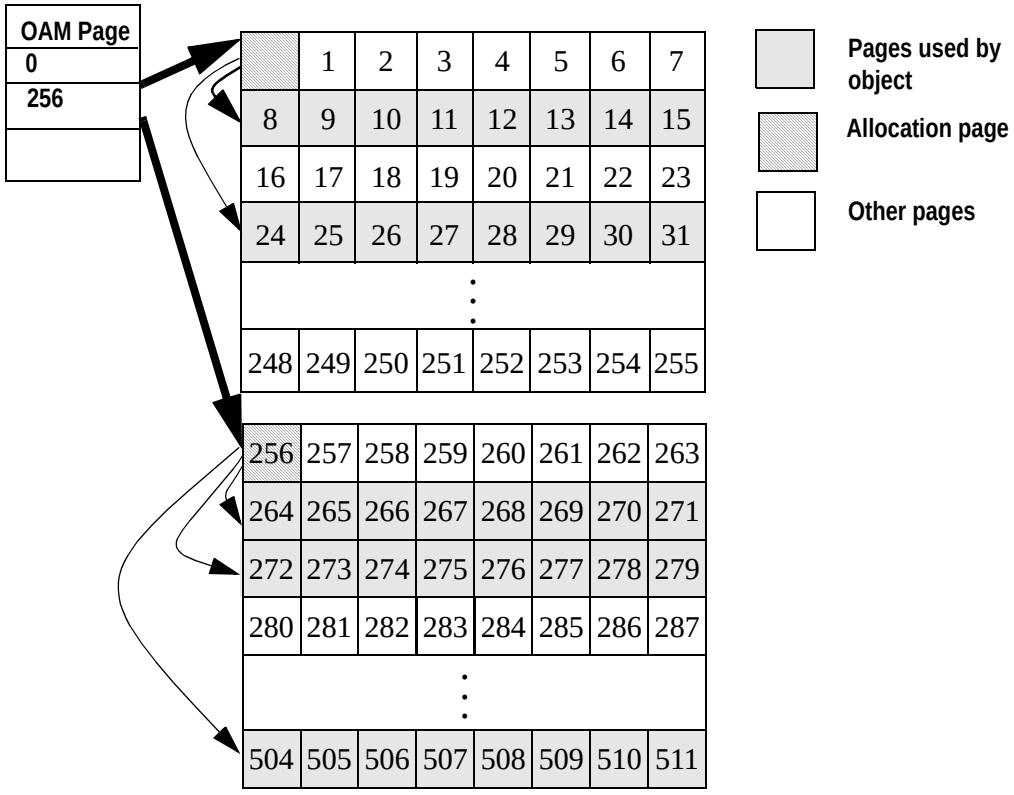
Tables that use data-only locking do not have page chains like allpages-locked tables. To perform a table scan on a data-only-locked table, Adaptive Server:

- Reads the OAM (object allocation map) page(s) for the table
- Uses the pointers on the OAM page to access the allocation pages
- Uses the pointers on the allocation pages to locate the extents used by the table
- Performs either large I/O or 2K I/O on the pages in the extent

The total cost of a table scan on a data-only-locked table includes the logical and physical I/O for all pages in the table, plus the cost of logical and physical I/O for the OAM and allocation pages.

Figure 20-1 shows the pointers from OAM pages to allocation pages and from allocation pages to extents.

***Figure 20-1: Sequence of pointers for OAM scans***




The formula for computing the cost of an OAM scan with 2K I/O is:

```
OAM Scan Cost =      (OAM_alloc_pages + Num_pages) * 18
              +  (OAM_alloc_pages + Num_pages)* 2
```

When large I/O can be used, the optimizer adds the cost of performing 2K I/O for the pages in the first extent of each allocation unit to the cost of performing 16K I/O on the pages in regular extents. The number of physical I/Os is the number of pages in the table, modified by a cluster adjustment that is based on the data page cluster ratio for the table.

See “How cluster ratios affect large I/O estimates” on page 433 for more information on cluster ratios.

Logical I/O costs are one I/O per page in the table, plus the logical I/O cost of reading the OAM and allocation pages. The formula for computing the cost of an OAM scan with large I/O is:

```
OAM Scan Cost =      OAM_alloc_pages * 18
                   + Pages in 1st extent * 18
                   + Pages in other extents / Pages per IO
  * Cluster adjustment * 18
                   + OAM_alloc_pages * 2
                   + Pages in table * 2
```

optdiag reports the number of pages for each of the needed values.

When a data-only-locked table contains forwarded rows, the I/O cost of reading the forwarded rows is added to the logical and physical I/O for a table scan.

See "Allpages-locked heap tables" on page 156 for more information on row forwarding.

# From rows to pages

When the optimizer costs the use of an index to resolve a query, it first estimates the number of qualifying rows, and then estimates the number of pages that need to be read.

The examples in Chapter 17, "Adaptive Server Optimizer," show how Adaptive Server estimates the number of rows for a search argument or join using statistics. Once the number of rows has been estimated, the optimizer estimates the number of data pages and index leaf pages that need to be read:

- For tables, the optimizer divides the number of rows in the table by the number of pages to determine the average number of rows per data page.
- To estimate the average number of rows per page on the leaf level of an index, the optimizer divides the number of rows in the table by the number of leaf pages in the index.

After the number of pages is estimated, data page and index page cluster ratios are used to adjust the page estimates for queries using large I/O, and data row cluster ratios are used to estimate the number of data pages for queries using noncovering indexes.

## How cluster ratios affect large I/O estimates

When clustering is high, large I/O is effective. As the cluster ratios decline, effectiveness of large I/O drops rapidly. To refine I/O estimates, the optimizer uses a set of cluster ratios:

- For a table, the data page cluster ratio measures the packing and sequencing of pages on extents.
- For an index, the data page cluster ratio measures the effectiveness of large I/O for accessing the table using this index.
- The index page cluster ratio measures the packing and sequencing of leaf-level index pages on index extents.

**Note** The data row cluster ratio, another cluster ratio used by query optimization, is used to cost the number of data pages that need to be accessed during scans using a particular index. It is not used in large I/O costing.

optdiag displays the cluster ratios for tables and indexes.

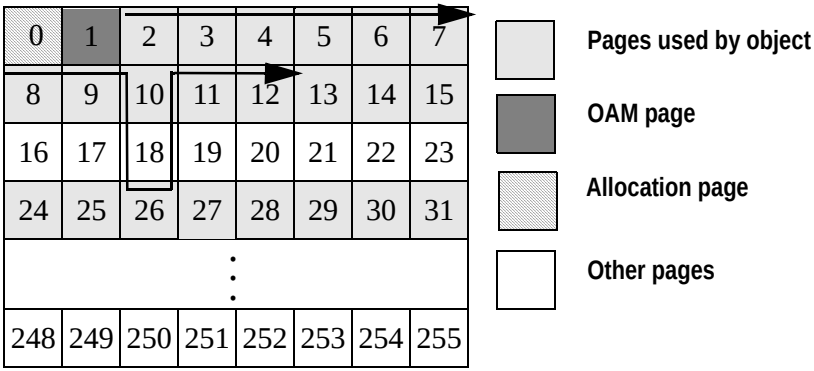
### Data page cluster ratio

The data page cluster ratio for a table measures the effectiveness of large I/O for table scans. Its use is slightly different depending on the locking scheme.

### On allpages-locked tables

For allpages-locked tables, a table scan or a scan that uses a clustered index to scan many pages follows the next-page pointers on each data page. Immediately after the clustered index is created, the data page cluster ratio is 1.0, and pages are ordered by page number on the extents. However, after updates and page splits, the page chain can be fragmented across the page chain, as shown in Figure 20-2, where page 10 has been split; the page pointers point from page 10 to page 26 in another extent, then to page 11.

***Figure 20-2: Page chain crossing extents in an allpages-locked table***




The data page cluster ratio for an allpages-locked table measures the effectiveness of large I/O for both table scans and clustered index scans.

### On data-only-locked tables

For data-only-locked tables, the data page cluster ratio measures how well the pages are packed on the extents. A cluster ratio of 1.0 indicates complete packing of extents, with the page chain ordered. If extents contain unused pages, the data page cluster ratio is less than 1.0.

optdiag reports two data page cluster ratios for data-only-locked tables with clustered indexes. The value reported for the table is used for table scans. The value reported for the clustered index is used for scans using the index.

### Index page cluster ratio

The index page cluster ratio measures the packing and sequencing of index leaf pages on extents for nonclustered indexes and clustered indexes on data-only-locked tables. For queries that need to read more than one leaf page, the leaf level of the index is scanned using next-page or previous-page pointers. If many leaf rows need to be read, 16K I/O can be used on the leaf pages to read one extent at a time. The index page cluster ratio measures fragmentation of the page chain for the leaf level of the index.

## Evaluating the cost of index access

When a query has search arguments on useful indexes, the query accesses only the index pages and data pages that contain rows that match the search arguments. Adaptive Server compares the total cost of index and data page I/O to the cost of performing a table scan, and uses the cheapest method.

### Query that returns a single row

A query that returns a single row using an index performs one I/O for each index level plus one read for the data page. The optimizer estimates the total cost as one physical I/O and one logical I/O for each index page and the data page. The cost for a point query is:

```
Point query cost   = (Number of index levels + data page) * 18
+ (Number of index levels + data page) * 2
```

optdiag output displays the number of index levels.

The root page and intermediate pages of frequently used indexes are often found in cache. In that case, actual physical I/O is reduced by one or two reads.

### Query that returns many rows

A query that returns many rows may be optimized very differently, depending on the type of index and the number of rows to be returned. Some examples are:

- Queries with search arguments that match many values, such as:

```
select title, price
from titles
where pub_id = "P099"
```

- Range queries, such as:

```
select title, price
from titles
where price between $20 and $50
```

For queries that return a large number of rows using the leading key of the index, clustered indexes and covering nonclustered indexes are very efficient:

- If the table uses allpages locking, and has a clustered index on the search arguments, the index is used to position the scan on the first qualifying row. The remaining qualifying rows are read by scanning forward on the data pages.
- If a nonclustered index or the clustered index on a data-only-locked table covers the query, the index is used to position the scan at the first qualifying row on the index leaf page, and the remaining qualifying rows are read by scanning forward on the leaf pages of the index.

If the index does not cover the query, using a clustered index on a data-only-locked table or a nonclustered index requires accessing the data page for each index row that matches the search arguments on the index. The matching rows may be scattered across many data pages, or they could be located on a very small number of pages, particularly if the index is a clustered index on a data-only-locked table. The optimizer uses data row cluster ratios to estimate how many physical and logical I/Os are required to read all of the qualifying data pages.

## Range queries using clustered indexes (allpages locking)

To estimate the number of physical I/Os required for a range query using a clustered index on an allpages-locked table, the optimizer adds the physical and logical I/O for each index level and the physical and logical I/O of reading the needed data pages. Since data pages are read in order following the page chain, the cluster adjustment helps estimate the effectiveness of large I/O. The formula is:

Data pages = Number of qualified rows / Data rows per page

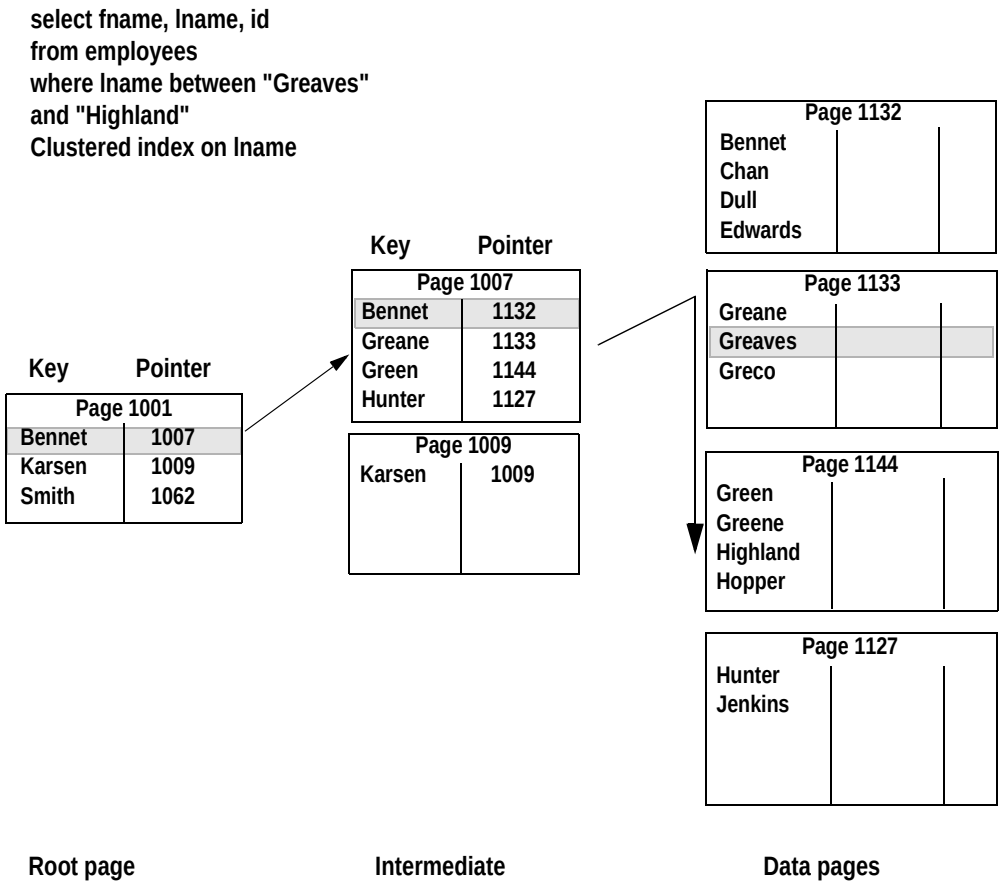
```
Range query cost = Number of index levels * 18
  + Data pages/pages per IO * Cluster adjustment * 18
  + Number of index levels * 2
  + Data pages * 2
```

If a query returns 500 rows, and the table has 10 rows per page, the query needs to read 50 data pages, plus one index page for each index level. If the query uses 2K I/O, it requires 50 I/Os for the data pages. If the query uses 16K I/O, these 50 data pages require 7 I/Os.

The cluster adjustment uses the data page cluster ratio to refine the estimate of large I/O for the table, based on how fragmented the data page storage has become on the table's extents.

Figure 20-3 shows how a range query using a clustered index positions the search on the first matching row on the data pages. The next-page pointers are used to scan forward on the data pages until a nonmatching row is encountered.

*Figure 20-3: Range query on the clustered index of an allpages-locked table*




## Range queries with covering indexes

Range queries using covering indexes perform very well because:

- The index is used to position the search at the first qualifying row on the index leaf level.
- Each index page contains more rows than corresponding data rows, so fewer pages need to be read.

- Index pages tend to remain in cache longer than data pages, so fewer physical I/Os are needed.
- If the cache used by the index is configured for large I/O, up to 8 leaf-level pages can be read per I/O.
- The data pages do not have to be accessed.

Both nonclustered indexes and clustered indexes on data-only-locked tables have a leaf level above the data level, so they can provide index covering.

The cost of using a covering index is determined by:

- The number of non-leaf index levels
- The number of rows that the query returns
- The number of rows per page on the leaf level of the index
- The number of leaf pages read per I/O
- The index page cluster ratio, used to adjust large I/O estimates when the index pages are not stored consecutively on the extents

This formula shows the costs:

```
Leaf pages = Number of qualified rows / Leaf level rows per page

Covered scan cost =Number of index levels * 18
    +(Leaf pages /Pages per IO) * Cluster adjustment * 18
    +Number of index levels * 2
    +Leaf pages * 2
```

For example, if a query needs to read 1,200 leaf pages, and there are 40 rows per leaf-level page, the query needs to read 30 leaf-level pages. If large I/O can be used, this requires 4 I/Os. If inserts have caused page splits on the index leaf-level, the cluster adjustment increases the estimated number of large I/Os.

## Range queries with noncovering indexes

When a nonclustered index or a clustered index on a data-only-locked table does not cover the query, Adaptive Server:

- Uses the index to locate the first qualifying row at the leaf level of the nonclustered index

- Follows the pointer to the data page for that index, and reads the page
- Finds the next row on the index page, and locates its data page, and continues this process until all matching keys have been used

For each subsequent key, the data row could be on the same page as the row for the previous key, or the data row may be on a different page in the table. The clustering of key values for each index is measured by a value called the *data row cluster ratio*. The data row cluster ratio is applied to estimate the number of logical and physical I/Os.

When the data row cluster ratio is 1.0, clustering is very high. High cluster ratios are always seen immediately after creating a clustered index; cluster ratios are 1.00000 or .999997, for example. Rows on the data pages are stored the same order as the rows in the index. The number of logical and physical I/Os needed for the data pages is (basically) the number of rows to be returned, divided by the number of rows per page. For a table with 10 rows per page, a query that needs to return 500 rows needs to read 50 pages if the data row cluster ratio is 1.

When the data row cluster ratio is extremely low, the data rows are scattered on data pages with no relationship to the ordering of the keys. Nonclustered indexes often have low data row cluster ratios, since there is no relationship between the ordering of the index keys and the ordering of the data rows on data pages. When the data row cluster ratio is 0, or close to 0, the number of physical and logical I/Os required could be as much as 1 data page I/O for each row to be returned. A query that needs to return 500 rows needs to read 500 pages, or nearly 500 pages, if the data row cluster ratio is near 0 and the rows are widely scattered on the data pages. In a huge table, this still provides good performance, but in a table with less than 500 pages, the optimizer chooses the cheaper alternative – a table scan.

The size of the data cache is also used in calculating the physical I/O. If the data row cluster ratio is very low, and the cache is small, pages may be flushed from cache before they can be reused. If the cache is large, the optimizer estimates that some pages will be found in cache.

## Result-set size and index use

A range query that returns a small number of rows performs well with the index, however, range queries that return a large number of rows may not use the index—it may be more expensive to perform the logical and physical I/O for a large number of index pages plus a large number of data pages. The lower the data row cluster ratio, the more expensive it is to use the index.

At the leaf level of a nonclustered index or a clustered index on a data-only-locked table, the keys are stored sequentially. For a search argument on a value that matches 100 rows, the rows on the index leaf level fit on perhaps one or two index pages. The actual data rows might all be on different data pages. The following queries show how different data row cluster ratios affect I/O estimates. The authors table uses datarows locking, and has these indexes:

- A clustered index on au_lname
- A nonclustered index on state

Each of these queries returns about 100 rows:

```
select au_lname, phone
from authors
where au_lname like "E%"
select au_id, au_lname, phone
from authors
where state = "NC"
```

The following table shows the data row cluster ratio for each index, and the optimizer's estimate of the number of rows to be returned and the number of pages required.

| SARG on | Data row cluster ratio | Row estimate | Page estimate | Data I/O size |
|---|---|---|---|---|
| au_lname | .999789 | 101 | 8 | 16K |
| state | .232539 | 103 | 83 | 2K |

The basic information on the table is:

- The table has 262 pages.
- There are 19 rows per data page in the table.

While each of the queries has its search clauses in valid search-argument form, and each of the clauses matches an index, only the first query uses the index: for the other query, a table scan is cheaper than using the index. With 262 pages, the cost of the table scan is:

```
Table scan cost =      (262 /8) = 37 * 18     =666
                  +    262 * 2                =524
                                              ____
                                              1190
```

### Closer look at the Search Argument costing

Looking more closely at the tables, cluster ratios, and search arguments explains why the table scan is chosen:

- The estimate for the clustered index on au_lname includes just 8 physical I/Os:
  - 6 I/Os (using 16K I/O) on the data pages, because the data row cluster ratio indicates very high clustering.
  - 2 I/Os for the index pages (there are 128 rows per leaf page); 16K I/O is also used for the index leaf pages.
- The query using the search argument on state has to read many more data pages, since the data row cluster ratio is low. The optimizer chooses 2K I/O on the data pages. 83 physical I/Os is more than double the physical I/O required for a table scan (using 16K I/O).

### Costing for noncovering index scans

The basic formula for estimating I/O for queries accessing the data through a noncovering index is:

```
Leaf pages =      Number of qualified rows / Leaf level rows per page

Data pages =      Number of qualifying rows * Data row cluster adjustment

Scan cost =       Number of nonleaf index levels * 18
                + (Leaf pages / Pages per IO) * Data page cluster adjustment * 18
                + (Data pages / Pages per IO) * Data page cluster adjustment * 18
                + Number of nonleaf index levels * 18
                + Leaf pages * 2
                + Number of qualifying rows * Data row cluster adjustment * 2
```

### Costing for forwarded rows

If a data-only-locked table has forwarded rows, the cost of the extra I/O for accessing forwarded rows is added for noncovered index scans. The cost is computed by multiplying the number of forwarded rows in the table and the percent of the rows from the table that to be returned by the query. The added cost is:

```
Forwarded row cost = % of rows returned * Number of forwarded rows in the table
```

## Costing for queries using *order by*

Queries that perform sorts for order by may create and sort, or they may be able to use the index to return rows by relying on the index ordering. For example, the optimizer chooses one of these access methods for a query with an order by clause:

- With no useful search arguments – use a table scan, followed by sorting the worktable.

- With selective search argument or join on an index that does not match the order by clause – use an index scan, followed by sorting the worktable.
- With a search argument or join on an index that matches the order by clause – an index scan using this index, with no worktable or sort.

Sorts are always required for result sets when the columns in the result set are a superset of the index keys. For example, if the index on authors includes au_fname and au_lname, and the order by clause also includes the au_id, the query requires a sort.

If there are search arguments on indexes that match the order by clause, and other search arguments on indexes that do not support the required ordering, the optimizer costs both access methods. If the worktable and sort is required, the cost of performing the I/O for these operations is added to the cost of the index scan. If an index is potentially useful to help avoid the sort, dbcc traceon(302) prints a message while the search or join argument costing takes place.

See “Sort avert messages” on page 881 for more information.

Besides the availability of indexes, two major factors determine whether the index is considered:

- The order by clause must specify a prefix subset of the index keys.
- The order by clause and the index must have compatible ascending/descending key ordering.

## Prefix subset and sorts

For a query to use an index to avoid a sort step, the keys specified in the order by clause must be a prefix subset of the index keys. For example, if the index specifies the keys as A, B, C, D:

- The following order by clauses can use the index:
  - A
  - A, B
  - A, B, C
  - A, B, C, D
- And other set of columns cannot use the index. For example, these are not prefix subsets:

- A, C
- B, C, D

## Key ordering and sorts

Both order by clauses and commands that create indexes can use the asc or desc (ascending or descending) ordering qualifications:

- For index creation, the asc and desc qualifications specify the order in which keys are to be stored in the index.
- In the order by clause, the ordering qualifications specify the order in which the columns are to be returned in the output.

To avoid a sort when using a specific index, the asc or desc qualifications in the order by clause must either be exactly the same as those used to create the index, or must be exactly the opposite.

### Specifying ascending or descending order for index keys

Queries that use a mix of ascending and descending order in an order by clause do not perform a separate sort step if the index was created using the same mix of ascending and descending order as that specified in the order by clause, or if the index order is the reverse of the order specified in the order by clause. Indexes are scanned forward or backward, following the page chain pointers at the leaf level of the index.

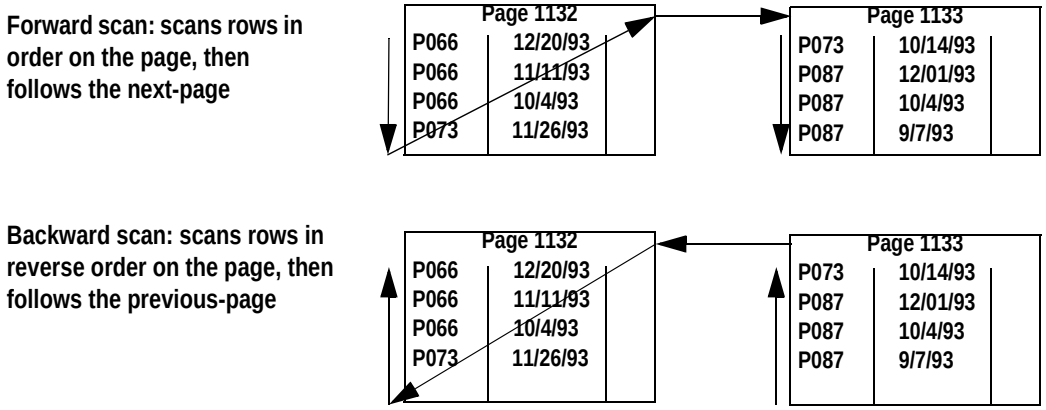
For example, this command creates an index on the titles table with pub_id ascending and pubdate descending:

```
create index pub_ix
    on titles (pub_id asc, pubdate desc)
```

The rows are ordered on the pages as shown in Figure 20-4. When the ascending and descending order in the query matches the index creation order, the result is a forward scan, starting at the beginning of the index or at the first qualifying row, returning the rows in order from each page, and following the next-page pointers to read subsequent pages.

If the ordering in the query is the exact opposite of the index creation order, the result is a backward scan, starting at the last page of the index or the page containing the last qualifying row, returning rows in backward order from each page, and following previous page pointers.

***Figure 20-4: Forward and backward scans on an index***




The following query using the index shown in Figure 20-4 performs a forward scan:

```
select *
from titles
order by pub_id asc, pubdate desc
```

This query using the index shown in Figure 20-4 performs a backward scan:

```
select *
from titles
order by pub_id desc, pubdate asc
```

For the following two queries on the same table, the plan requires a sort step, since the order by clauses do not match the ordering specified for the index:

```
select *
from titles
order by pub_id desc, pubdate desc
select *
from titles
order by pub_id asc, pubdate asc
```

---

Note Parallel sort operations are optimized very differently for partitioned tables. See Chapter 24, “Parallel Sorting,” for more information.

---

## How the optimizer costs sort operations

When Adaptive Server optimizes queries that require sorts:

- It computes the cost of using an index that matches the required sort order, if such an index exists.
- It computes the physical and logical I/O cost of creating a worktable and performing the sort for every index where the index order does not match the sort order. It computes the physical and logical I/O cost of performing a table scan, creating a worktable, and performing the sort.

Adding the cost of creating and sorting the worktable to the cost of index access and the cost of creating and sorting the worktable favors the use of an index that supports the order by clause. However, when comparing indexes that are very selective, but not ordered, versus indexes that are ordered, but not selective:

- Access costs are low for the more selective index, and so are sort costs.
- Access costs are high for the less selective index, and may exceed the cost of access using the more selective index and sort.

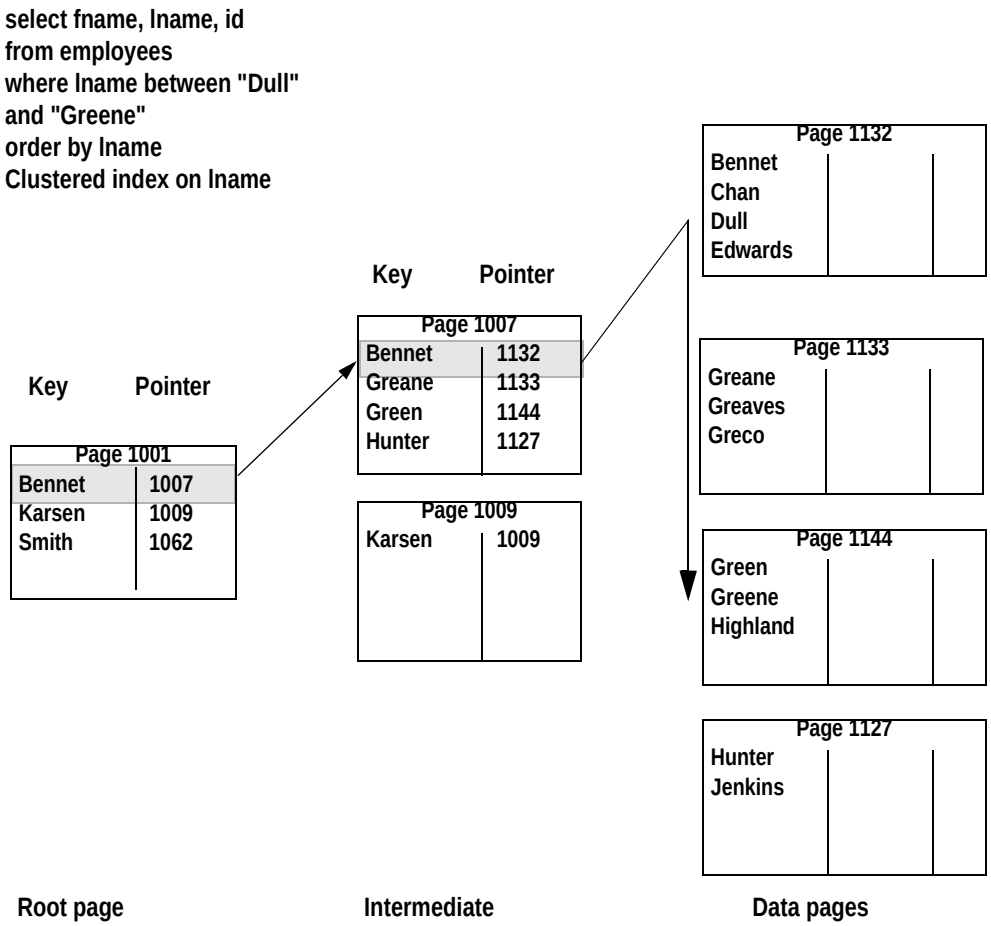
## Allpages-locked tables with clustered indexes

For allpages-locked tables with clustered indexes, order by queries that match the index keys are efficient if:

- There is also a search argument that uses the index, the index key positions the search on the data page for first qualifying row.
- The scan follows the next-page pointers until all qualifying rows have been found.
- No sort is needed.

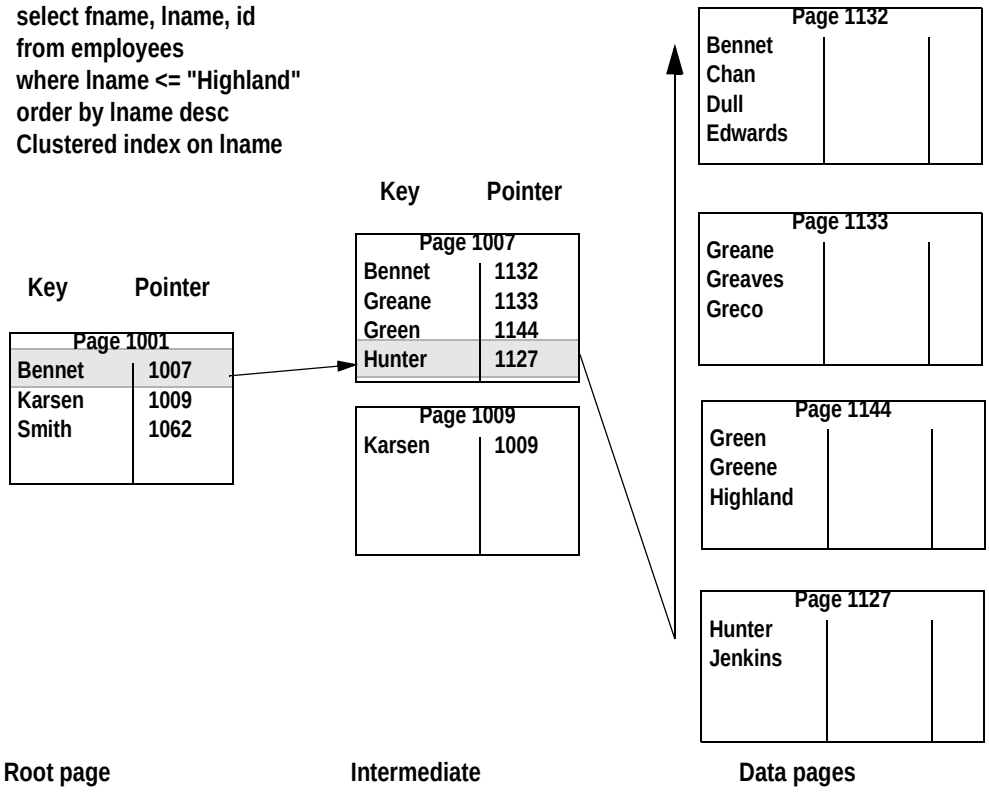
In Figure 20-5, the index was created in ascending order, and the order by clause does not specify the order, so ascending is used by default.

***Figure 20-5: An order by query using a clustered index, allpages locking***




Queries requiring descending sort order (for example, order by title_id desc) can avoid sorting by scanning pages in reverse order. If the entire table is needed for a query without a where clause, Adaptive Server follows the index pointers to the last page, and then scans backward using the previous page pointers. If the where clause includes an index key, the index is used to position the search, and then the pages are scanned backward, as shown in Figure 20-6.

***Figure 20-6: An order by desc query using a clustered index***




## Sorts when index covers the query

When an index covers the query and the order by columns form a prefix subset of the index keys, the rows are returned directly from the nonclustered index leaf pages. If the columns do not form a prefix subset of the index keys, a worktable is created and sorted.

With a nonclustered index on au_lname, au_fname, au_id of the authors table, this query can return the data directly from the leaf pages:

```
select au_id, au_lname
from authors
order by au_lname, au_fname
```

## Sorts and noncovering indexes

With a noncovering index, Adaptive Server determines whether using the index that supports the ordering requirements is cheaper than performing a table scan or using a more selective index, and then inserting rows into a worktable and sorting the data. The cost of using the index depends on the number of rows and the data row cluster ratio.

### Backward scans and joins

If two or more tables are being joined, and the order by clause specifies descending order for index keys on the joined tables, any of the tables and indexes involved can be scanned with a backward scan to avoid the worktable and sort costs. If all the columns for one table are in ascending order, and the columns for the other tables are in descending order, the first table is scanned in ascending order and the others in descending order.

### Deadlocks and descending scans

Descending scans may deadlock with queries performing update operations using ascending scans and with queries performing page splits and shrinks, except when the backward scans are performed at transaction isolation level 0.

The allow backward scans configuration parameter controls whether the optimizer uses the backward scan strategy. The default value of 1 allows descending scans.

See the *System Administration Guide* for more information on this parameter.

Also, see “Index scans” on page 961 for information on the number of ascending and descending scans performed and “Deadlocks by lock type” on page 969 for information on detecting deadlocks.

# Access Methods and Costing for *or* and *in* Clauses

When a query on a single table contains or clauses or an in (values_list) clause, it can be optimized in different ways, depending on the presence of indexes, the selectivity of the search arguments, the existence of other search arguments, and whether or not the clauses might return duplicate rows.

## *or* syntax

or clauses take one of the following forms:

```
where column_name1 = <value>
   or column_name1 = <value>
   ...
```

or:

```
where column_name1 = <value>
   or column_name2 = <value>
   ...
```

## *in (values_list)* converts to *or* processing

Preprocessing converts in lists to or clauses, so this query:

```
select title_id, price
    from titles
    where title_id in ("PS1372", "PS2091","PS2106")
```

becomes:

```
select title_id, price
    from titles
    where title_id = "PS1372"
        or title_id = "PS2091"
        or title_id = "PS2106"
```

## Methods for processing *or* clauses

A single-table query including or clauses is a union of more than one query. Although some rows may match more than one of the conditions, each row must be returned only once. Depending on indexes and query clauses, or queries can be resolved by one of these methods:

- If any of the clauses linked by or is not indexed, the query must use a table scan. If there is an index on type, but no index on advance, this query performs a table scan:

  ```
  select title_id, price
  from titles
  where type = "business" or advance > 10000
  ```

- If there is a possibility that one or more of the or clauses could match values in the same row, the query is resolved using the ***OR strategy***, also known as using a ***dynamic index***. The OR strategy selects the row IDs for matching rows into a worktable, and sorts the worktable to remove duplicate row IDs. For example, there can be rows for which both of these conditions are true:

  ```
  select title_id
  from titles
  where pub_id = "P076" or type > "business"
  ```

  If there is an index on pub_id, and another on type, the OR strategy can be used.

  See “Dynamic index (OR strategy)” on page 454 for more information.

  **Note** The *OR Strategy* (multiple matching index scans) is only considered for equality predicates. It is disqualified for range predicates even if meeting other conditions. As an example, when a select statement contains the following:

  ```
  where bar between 1 and 5
  or bar between 10 and 15
  ```

  This will not be considered for the *OR Strategy.*

- If there is no possibility that the or clauses can select the same row, the query can be resolved with multiple matching index scans, also known as the ***special OR strategy***. The special OR strategy does not require a worktable and sort. The or clauses in this query cannot select the same row twice:

```
select title_id, price
from titles
where pub_id = "P076" or pub_id = "P087"
```

With an index on pub_id, this query can be resolved using two matching index scans.

See "Multiple matching index scans (special OR strategy)" on page 456 for more information.

- The costs of index access for each or clause are added together, and the cost of the sort, if required. If sum of these costs is greater than a table scan, the table scan is chosen. For example, this query uses a table scan if the total cost of all of the indexed scans on pub_id is greater than the table scan:

  ```
  select title_id, price
  from titles
  where pub_id in ("P095", "P099", "P128", "P220",
  "P411", "P445", "P580", "P988")
  ```

- If the query contains additional search arguments on indexed columns, predicate transformation may add search arguments that can be optimized, adding alternative optimization options. The cost of using all alternative access methods is compared, and the cheapest alternative is selected. This query contains a search argument on type as well as clauses linked with or:

  ```
  select title_id, type, price from titles
  where type = "business"
  and (pub_id = "P076" or pubdate > "12/1/93")
  ```

  With a separate index on each search argument, the optimizer uses the least expensive access method:

  - The index on type
  - The OR strategy on pub_id and pubdate

### When table scans are used for *or* queries

A query with or clauses or an in (values_list) uses a table scan if either of these conditions is true:

- The cost of all the index accesses is greater than the cost of a table scan, or
- At least one of the columns is not indexed, so the only way to resolve the query conditions is to perform a table scan.

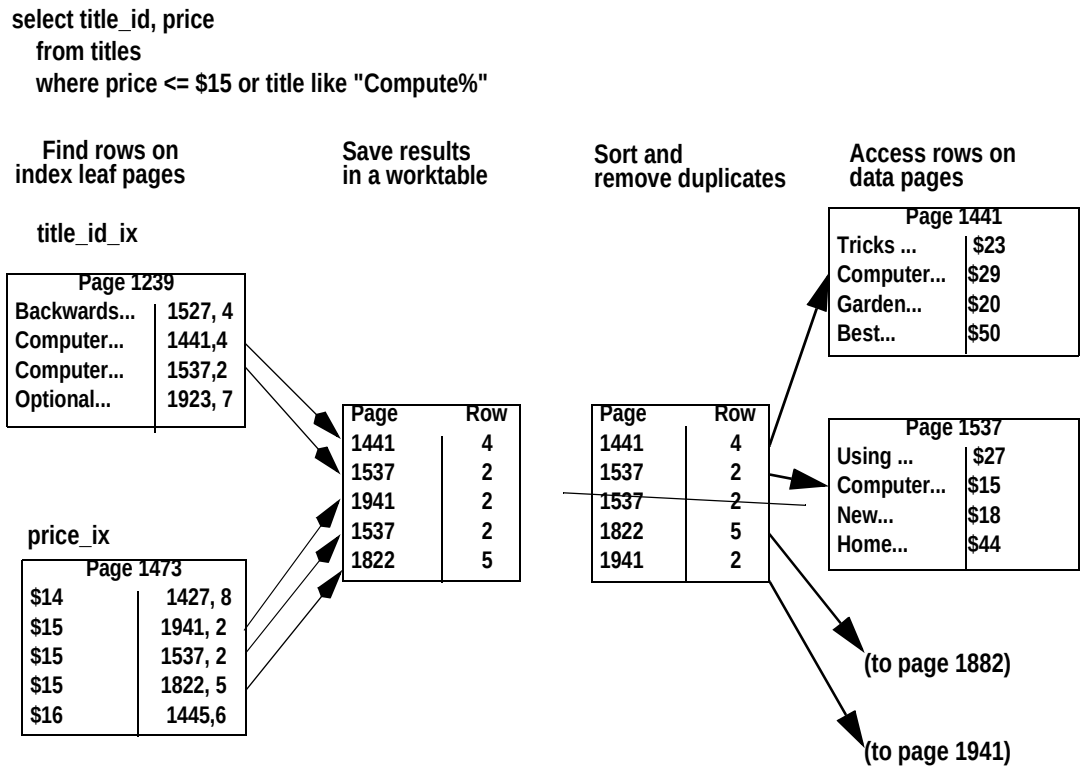
## Dynamic index (OR strategy)

If the query uses the OR strategy because the query could return duplicate rows, the appropriate indexes are used to retrieve the row IDs for rows that satisfy each or clause. The row IDs for each or clause are stored in a worktable. Since the worktable contains only row IDs, it is called a "dynamic index." Adaptive Server then sorts the worktable to remove the duplicate row IDs. The row IDs are used to retrieve the rows from the base tables. The total cost of the query includes:

- The sum of the index accesses, that is, for each or clause, the cost of using the index to access the row IDs on the leaf pages of the index (or on the data pages, for a clustered index on an allpages-locked table)
- The cost of reading the worktable and performing the sort
- The cost of using the row IDs to access the data pages

Figure 20-7 illustrates the process of building and sorting a dynamic index for an or query on two different columns.

***Figure 20-7: Resolving or queries using the OR strategy***




As shown in Figure 20-7, the optimizer can choose to use a different index for each clause.

showplan displays “Using Dynamic Index” and “Positioning by Row IDentifier (RID)” when the OR strategy is used.

See “Dynamic index message (OR strategy)” on page 807 for more information.

Queries in cursors cannot use the OR strategy, but must perform a table scan. However, queries in cursors can use the multiple matching index scans strategy.

Locking during queries that use the OR strategy depends on the locking scheme of the table.

### Multiple matching index scans (special OR strategy)

Adaptive Server uses multiple matching index scans when the or clauses are on the same table, and there is no possibility that the or clauses will return duplicate rows. For example, this query cannot return any duplicate rows:

```
select title
    from titles
    where title_id in ("T6650", "T95065", "T11365")
```

This query can be resolved using multiple matching index scans, using the index on title_id. The total cost of the query is the sum of the multiple index accesses performed. If the index on title_id has 3 levels, each or clause requires 3 index reads, plus one data page read, so the total cost for each clause is 4 logical and 4 physical I/Os, and the total query cost is estimated to be 12 logical and 12 physical I/Os.

The optimizer determines which index to use for each or clause or value in the in (values_list) clause by costing each clause or value separately. If each column named in a clause is indexed, a different index can be used for each clause or value. showplan displays the message "Using *N* Matching Index Scans" when the special OR strategy is used.

See "Matching index scans message" on page 806.

## How aggregates are optimized

Aggregates are processed in two steps:

- First, appropriate indexes are used to retrieve the appropriate rows, or a table scan is performed. For vector (grouped) aggregates, the results are placed in a worktable. For scalar aggregates, results are computed in a variable in memory.
- Second, the worktable is scanned to return the results for vector aggregates, or the results are returned from the internal variable.

Vector aggregates can use a covering composite index on the aggregated column and the grouping column, if any, rather than performing table scans. For example, if the titles table has a nonclustered index on type, price, the following query retrieves its results by scanning the leaf level of the nonclustered index:

```
select type, avg(price)
    from titles
    group by type
```

Scalar aggregates can also use covering indexes to reduce I/O. For example, the following query can use the index on type, price:

```
select min(price)
    from titles
```

Table 20-1 shows some of the access methods that the optimizer can choose for queries with aggregates when there is no where, having or group by clause in the query.

***Table 20-1: Special access methods for aggregates***

| Aggregate | Index description | Access method |
|---|---|---|
| min | Scalar aggregate is leading column | Use first the value on the root page of the index. |
| max | Clustered index on an allpages-locked table | Follow the last pointer on root page and intermediate pages to data page, and return the last value. |
| | Clustered index on a data-only-locked table<br>Any nonclustered index | Follow last pointer on root page and intermediate pages to leaf page, and return the last value. |
| count(*) | Nonclustered index or clustered index on a data-only-locked table | Count all rows in the leaf level of the index with the smallest number of pages. |
| count(*col_name*) | Covering nonclustered index, or covering clustered index on data-only-locked table | Count all non-null values in the leaf level of the smallest index containing the column name. |

## Combining *max* and *min* aggregates

When used separately, max and min aggregates on leading index columns use special processing if there is no where clause in the query:

- min aggregates retrieve the first value on the root page of the index, performing a single read to find the value.
- max aggregates follow the last entry on the last page at each index level until they reach the leaf level.

However, when min and max are used together, this optimization is not available. The entire leaf level of an index is scanned to locate the first and last values.

min and max optimizations are not applied if:

- The expression inside the max or min function is anything but a column. When *numeric_col* has a nonclustered index:
  - max(*numeric_col**2) contains an operation on a column, so the query performs a leaf-level scan of the index.
  - max(*numeric_col*)*2 uses max optimization, because the multiplication is performed on the result of the function.
- There is another aggregate in the query.
- There is a group by clause.

### Queries that use both *min* and *max*

If you have max and min aggregates that can be optimized, you should get much better performance by putting them in separate queries. For example, even if there is an index with price as the leading key, this query results in a full leaf-level scan of the index:

```
select max(price), min(price)
    from titles
```

When you separate them, Adaptive Server uses the index once for each of the two queries, rather than scanning the entire leaf level. This example shows two queries:

```
select max(price)
    from titles
select min(price)
    from titles
```

# How update operations are performed

Adaptive Server handles updates in different ways, depending on the changes being made to the data and the indexes used to locate the rows. The two major types of updates are **deferred updates** and **direct updates**. Adaptive Server performs direct updates whenever possible.

## Direct updates

Adaptive Server performs direct updates in a single pass:

- It locates the affected index and data rows.
- It writes the log records for the changes to the transaction log.
- It makes the changes to the data pages and any affected index pages.

There are three techniques for performing direct updates:

- In-place updates
- Cheap direct updates
- Expensive direct updates

Direct updates require less overhead than deferred updates and are generally faster, as they limit the number of log scans, reduce logging, save traversal of index B-trees (reducing lock contention), and save I/O because Adaptive Server does not have to refetch pages to perform modifications based on log records.

## In-place updates

Adaptive Server performs in-place updates whenever possible.

When Adaptive Server performs an in-place update, subsequent rows on the page are not moved; the row IDs remain the same and the pointers in the row offset table are not changed.

For an in-place update, the following requirements must be met:

- The row being changed cannot change its length.
- The column being updated cannot be the key, or part of the key, of a clustered index on an allpages-locked table. Because the rows in a clustered index on an allpages-locked table are stored in key order, a change to the key almost always means that the row location is changed.
- One or more indexes must be unique or must allow duplicates.
- The update statement satisfies the conditions listed in “Restrictions on update modes through joins” on page 465.
- The affected columns are not used for referential integrity.
- There cannot be a trigger on the column.
- The table cannot be replicated (via Replication Server).

An in-place update is the fastest type of update because it makes a single change to the data page. It changes all affected index entries by deleting the old index rows and inserting the new index row. In-place updates affect only indexes whose keys are changed by the update, since the page and row locations are not changed.

## Cheap direct updates

If Adaptive Server cannot perform an update in place, it tries to perform a cheap direct update—changing the row and rewriting it at the same offset on the page. Subsequent rows on the page are moved up or down so that the data remains contiguous on the page, but the row IDs remain the same. The pointers in the row offset table change to reflect the new locations.

A cheap direct update,must meet these requirements:

- The length of the data in the row is changed, but the row still fits on the same data page, or the row length is not changed, but there is a trigger on the table or the table is replicated.
- The column being updated cannot be the key, or part of the key, of a clustered index. Because Adaptive Server stores the rows of a clustered index in key order, a change to the key almost always means that the row location is changed.
- One or more indexes must be unique or must allow duplicates.
- The update statement satisfies the conditions listed in “Restrictions on update modes through joins” on page 465.
- The affected columns are not used for referential integrity.

Cheap direct updates are almost as fast as in-place updates. They require the same amount of I/O, but slightly more processing. Two changes are made to the data page (the row and the offset table). Any changed index keys are updated by deleting old values and inserting new values. Cheap direct updates affect only indexes whose keys are changed by the update, since the page and row ID are not changed.

## Expensive direct updates

If the data does not fit on the same page, Adaptive Server performs an expensive direct update, if possible. An expensive direct update deletes the data row, including all index entries, and then inserts the modified row and index entries.

Adaptive Server uses a table scan or an index to find the row in its original location and then deletes the row. If the table has a clustered index, Adaptive Server uses the index to determine the new location for the row; otherwise, Adaptive Server inserts the new row at the end of the heap.

An expensive direct updatemust meet these requirements:

- The length of a data row is changed so that the row no longer fits on the same data page, and the row is moved to a different page, or the update affects key columns for the clustered index.
- The index used to find the row is not changed by the update.
- The update statement satisfies the conditions listed in "Restrictions on update modes through joins" on page 465.
- The affected columns are not used for referential integrity.

An expensive direct update is the slowest type of direct update. The delete is performed on one data page, and the insert is performed on a different data page. All index entries must be updated, since the row location is changed.

## Deferred updates

Adaptive Server uses deferred updates when direct update conditions are not met. A deferred update is the slowest type of update.

In a deferred update, Adaptive Server:

- Locates the affected data rows, writing the log records for deferred delete and insert of the data pages as rows are located.
- Reads the log records for the transaction and performs the deletes on the data pages and any affected index rows.
- Reads the log records a second time, and performs all inserts on the data pages, and inserts any affected index rows.

### When deferred updates are required

Deferred updates are always required for:

- Updates that use self-joins
- Updates to columns used for self-referential integrity

- Updates to a table referenced in a correlated subquery

Deferred updates are also required when:

- The update moves a row to a new page while the table is being accessed via a table scan or a clustered index.
- Duplicate rows are not allowed in the table, and there is no unique index to prevent them.
- The index used to find the data row is not unique, and the row is moved because the update changes the clustered index key or because the new row does not fit on the page.

Deferred updates incur more overhead than direct updates because they require Adaptive Server to reread the transaction log to make the final changes to the data and indexes. This involves additional traversal of the index trees.

For example, if there is a clustered index on title, this query performs a deferred update:

```
update titles set title = "Portable C Software" where
title = "Designing Portable Software"
```

## Deferred index inserts

Adaptive Server performs deferred index updates when the update affects the index used to access the table or when the update affects columns in a unique index. In this type of update, Adaptive Server:

- Deletes the index entries in direct mode
- Updates the data page in direct mode, writing the deferred insert records for the index
- Reads the log records for the transaction and inserts the new values in the index in deferred mode

Deferred index insert mode must be used when the update changes the index used to find the row or when the update affects a unique index. A query must update a single, qualifying row only once—deferred index update mode ensures that a row is found only once during the index scan and that the query does not prematurely violate a uniqueness constraint.

The update in Figure 20-8 changes only the last name, but the index row is moved from one page to the next. To perform the update, Adaptive Server:

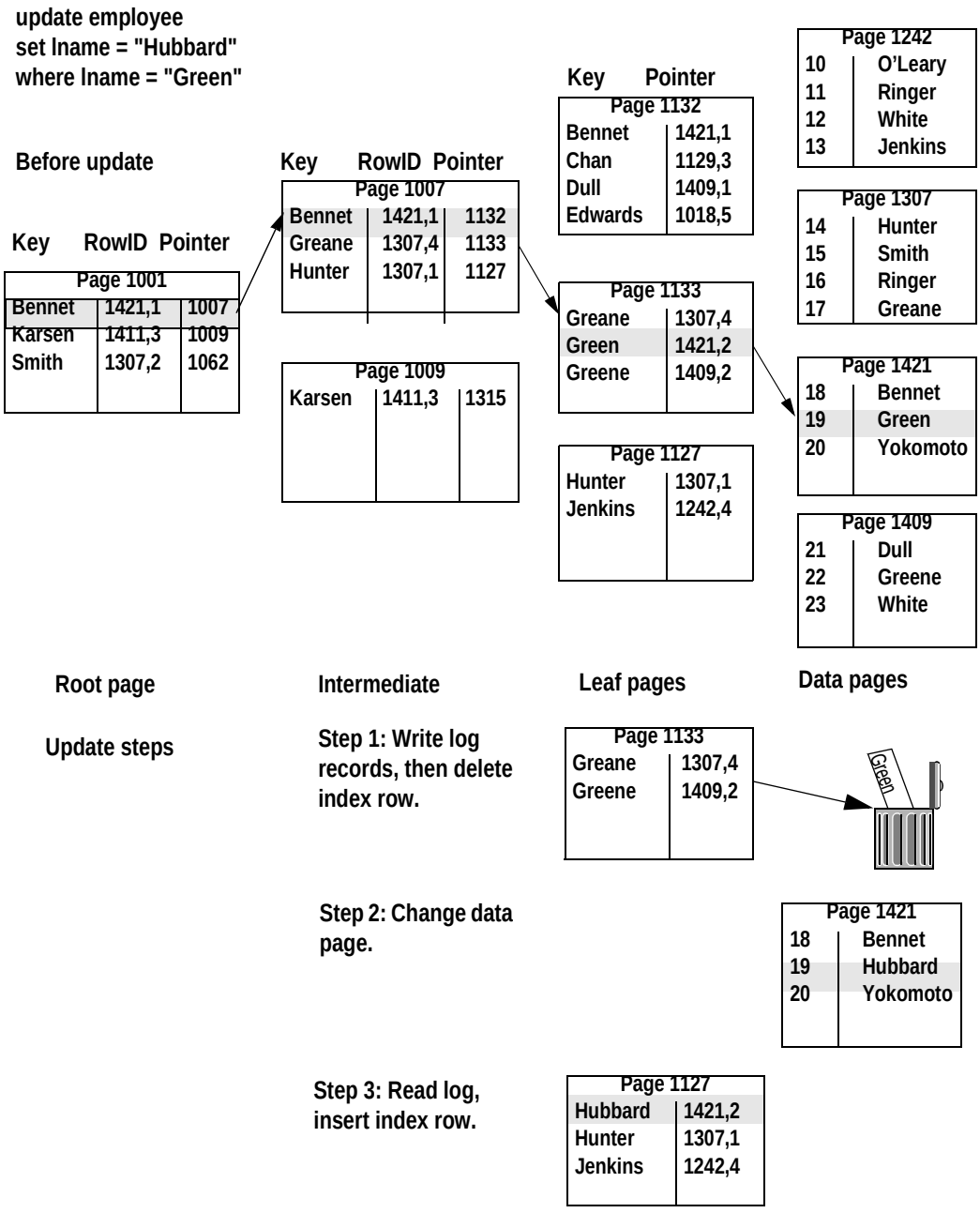
1 Reads index page 1133, deletes the index row for “Greene” from that page, and logs a deferred index scan record.

2 Changes “Green” to “Hubbard” on the data page in direct mode and continues the index scan to see if more rows need to be updated.

3 Inserts the new index row for “Hubbard” on page 1127.

Figure 20-8 shows the index and data pages prior to the deferred update operation, and the sequence in which the deferred update changes the data and index pages.

***Figure 20-8: Deferred index update***



Assume a similar update to the titles table:

```
update titles
set title = "Computer Phobic's Manual",
    advance = advance * 2
where title like "Computer Phob%"
```

This query shows a potential problem. If a scan of the nonclustered index on the title column found "Computer Phobia Manual," changed the title, and multiplied the advance by 2, and then found the new index row "Computer Phobic's Manual" and multiplied the advance by 2, the advance wold be very skewed against the reality.

A deferred index delete may be faster than an expensive direct update, or it may be substantially slower, depending on the number of log records that need to be scanned and whether the log pages are still in cache.

During deferred update of a data row, there can be a significant time interval between the delete of the index row and the insert of the new index row. During this interval, there is no index row corresponding to the data row. If a process scans the index during this interval at isolation level 0, it will not return the old or new value of the data row.

## Restrictions on update modes through joins

Updates and deletes that involve joins can be performed in direct, deferred_varcol, or deferred_index mode when the table being updated is the outermost table in the join order, or when it is preceded in the join order by tables where only a single row qualifies.

### Joins and subqueries in update and delete statements

The use of the from clause to perform joins in update and delete statements is a Transact-SQL extension to ANSI SQL. Subqueries in ANSI SQL form can be used in place of joins for some updates and deletes.

This example uses the from syntax to perform a join:

```
update t1 set t1.c1 = t1.c1 + 50
from t1, t2
where t1.c1 = t2.c1
and t2.c2 = 1
```

The following example shows the equivalent update using a subquery:

```
update t1 set c1 = c1 + 50
```

```
where t1.c1 in (select t2.c1
                from t2
                where t2.c2 = 1)
```

The update mode that is used for the join query depends on whether the updated table is the outermost query in the join order—if it is not the outermost table, the update is performed in deferred mode. The update that uses a subquery is always performed as a direct, deferred_varcol, or deferred_index update.

For a query that uses the from syntax and performs a deferred update due to the join order, use showplan and statistics io to determine whether rewriting the query using a subquery can improve performance. Not all queries using from can be rewritten to use subqueries.

### Deletes and updates in triggers versus referential integrity

Triggers that join user tables with the deleted or inserted tables are run in deferred mode. If you are using triggers solely to implement referential integrity, and not to cascade updates and deletes, then using declarative referential integrity in place of triggers may avoid the penalty of deferred updates in triggers.

## Optimizing updates

showplan messages provide information about whether an update is performed in direct mode or deferred mode. If a direct update is not possible, Adaptive Server updates the data row in deferred mode. There are times when the optimizer cannot know whether a direct update or a deferred update will be performed, so two showplan messages are provided:

- The “deferred_varcol” message shows that the update may change the length of the row because a variable-length column is being updated. If the updated row fits on the page, the update is performed in direct mode; if the update does not fit on the page, the update is performed in deferred mode.
- The “deferred_index” message indicates that the changes to the data pages and the deletes to the index pages are performed in direct mode, but the inserts to the index pages are performed in deferred mode.

These types of direct updates depend on information that is available only at runtime, since the page actually has to be fetched and examined to determine whether the row fits on the page.

## Designing for direct updates

When you design and code your applications, be aware of the differences that can cause deferred updates. Follow these guidelines to help avoid deferred updates:

- Create at least one unique index on the table to encourage more direct updates.
- Whenever possible, use nonkey columns in the where clause when updating a different key.
- If you do not use null values in your columns, declare them as not null in your create table statement.

## Effects of update types and indexes on update modes

Table 20-2 shows how indexes affect the update mode for three different types of updates. In all cases, duplicate rows are not allowed. For the indexed cases, the index is on title_id. The three types of updates are:

- Update of a variable-length key column:

```
update titles set title_id = value
    where title_id = "T1234"
```

- Update of a fixed-length nonkey column:

```
update titles set pub_date = value
        where title_id = "T1234"
```

- Update of a variable-length nonkey column:

```
update titles set notes = value
    where title_id = "T1234"
```

Table 20-2 shows how a unique index can promote a more efficient update mode than a nonunique index on the same key. Pay particular attention to the differences between direct and deferred in the shaded areas of the table. For example, with a unique clustered index, all of these updates can be performed in direct mode, but they must be performed in deferred mode if the index is nonunique.

For a table with a nonunique clustered index, a unique index on any other column in the table provides improved update performance. In some cases, you may want to add an IDENTITY column to a table in order to include the column as a key in an index that would otherwise be nonunique.

***Table 20-2: Effects of indexing on update mode***

| | Update To: | | |
|---|---|---|---|
| **Index** | **Variable-length key** | **Fixed-length column** | **Variable-length column** |
| No index | N/A | direct | deferred_varcol |
| Clustered, unique | direct | direct | direct |
| Clustered, not unique | deferred | deferred | deferred |
| Clustered, not unique, with a unique index on another column | deferred | direct | deferred_varcol |
| Nonclustered, unique | deferred_varcol | direct | direct |
| Nonclustered, not unique | deferred_varcol | direct | deferred_varcol |

If the key for an index is fixed length, the only difference in update modes from those shown in the table occurs for nonclustered indexes. For a nonclustered, nonunique index, the update mode is deferred_index for updates to the key. For a nonclustered, unique index, the update mode is direct for updates to the key.

If the length of varchar or varbinary is close to the maximum length, use char or binary instead. Each variable-length column adds row overhead and increases the possibility of deferred updates.

Using max_rows_per_page to reduce the number of rows allowed on a page increases direct updates, because an update that increases the length of a variable-length column may still fit on the same page.

For more information on using max_rows_per_page, see “Using max_rows_per_page on allpages-locked tables” on page 291.

## Using *sp_sysmon* while tuning updates

You can use showplan to determine whether an update is deferred or direct, but showplan does not give you detailed information about the type of deferred or direct update. Output from the sp_sysmon or Adaptive Server Monitor supplies detailed statistics about the types of updates performed during a sample interval.

Run sp_sysmon as you tune updates, and look for reduced numbers of deferred updates, reduced locking, and reduced I/O.

See “Transaction detail” on page 942 for more information.

CHAPTER 21

# Accessing Methods and Costing for Joins and Subqueries

This chapter introduces the methods that Adaptive Server uses to access rows in tables when more than one table is used in a query, and how the optimizer costs access.



In determining the cost of multitable queries, Adaptive Server uses many of the same formulas discussed in Chapter 20, “Access Methods and Query Costing for Single Tables.”

## Costing and optimizing joins

Joins extract information from two or more tables. In a two-table join, one table is treated as the outer table and the other table is treated as the inner table. Adaptive Server examines the outer table for rows that satisfy the query conditions. For each row in the outer table that qualifies, Adaptive Server then examines the inner table, looking at each row where the join columns match.

Optimizing join queries is extremely important for system performance, since relational databases make heavy use of joins. Queries that perform joins on several tables are especially critical to performance, as explained in the following sections.

In showplan output, the order of "FROM TABLE" messages indicates the order in which Adaptive Server chooses to join tables.

See "FROM TABLE message" on page 775 for an example that joins three tables. Some subqueries are also converted to joins.

See "Flattening in, any, and exists subqueries" on page 494.

## Processing

By default, Adaptive Server uses nested-loop joins, and also consider merge joins, if this feature is enabled at the server-wide or session level.

When merge joins are enabled, Adaptive Server can use either nested-loop joins or merge joins to process queries involving two or more tables. For each join, the optimizer costs both methods. For queries involving more than two tables, the optimizer examines query costs for merge joins and for nested-loops, and chooses the mix of merge and nested-loop joins that provides the cheapest query cost.

## Index density and joins

The optimizer uses a statistic called the **total density** to estimate the number of rows in a joined table that match a particular value during the join.

See "Density values and joins" on page 394 for more information.

The query optimizer uses the total density to estimate the number of rows that will be returned for each scan of the inner table of a join. For example, if the optimizer is considering a nested-loop join with a 250,000-row table, and the table has a density of .0001, the optimizer estimates that an average of 25 rows from the inner table match for each row that qualifies in the outer table.

optdiag reports the total density for each column for which statistics have been created. You can also see the total density used for joins in dbcc traceon(302) output.

### Multicolumn densities

Adaptive Server maintains the total density for each prefix subset of columns in a composite index. If two tables are being joined on multiple leading columns of a composite index, the optimizer uses the appropriate density for an index when estimating the cost of a join using that index. In a 10,000-row table with an index on seven columns, the entire seven-column key might have a density of 1/10,000, while the first column might have a density of only 1/2, indicating that it would return 5000 rows.

## Datatype mismatches and joins

One of the most common problems in optimizing joins on tables that have indexes is that the datatypes of the join columns are incompatible. When this occurs, one of the datatypes must be converted to the other, and an index can only be used for one side of the join.

See “Datatype mismatches and query optimization” on page 395 for more information.

## Join permutations

When you are joining four or fewer tables, Adaptive Server considers all possible permutations of join orders for the tables. However, due to the iterative nature of Adaptive Server’s optimizer, queries on more than four tables examine join order combinations in sets of two to four tables at a time. This grouping during join order costing is used because the number of permutations of join orders multiplies with each additional table, requiring lengthy computation time for large joins. The method the optimizer uses to determine join order has excellent results for most queries and requires much less CPU time than examining all permutations of all combinations.

If the number of tables in a join is greater than 25, Adaptive Server automatically reduces the number of tables considered at a time. Table 21-1 shows the default values.

***Table 21-1: Tables considered at a time during a join***

| Tables joined | Tables considered at a time |
|---|---|
| 4 – 25 | 4 |
| 26 – 37 | 3 |
| 38 – 50 | 2 |

The optimizer starts by considering the first two to four tables, and determining the best join order for those tables. It remembers the outer table from the best plan involving the tables it examined and eliminates that table from the set of tables. Then, it optimizes the best set of tables out of the remaining tables. It continues until only two to four tables remain, at which point it optimizes them.

For example, suppose you have a select statement with the following from clause:

```
from T1, T2, T3, T4, T5, T6
```

The optimizer looks at all possible sets of 4 tables taken from these 6 tables. The 15 possible combinations of all 6 tables are:

```
T1, T2, T3, T4
T1, T2, T3, T5
T1, T2, T3, T6
T1, T2, T4, T5
T1, T2, T4, T6
T1, T2, T5, T6
T1, T3, T4, T5
T1, T3, T4, T6
T1, T3, T5, T6
T1, T4, T5, T6
T2, T3, T4, T5
T2, T3, T4, T6
T2, T3, T5, T6
T2, T4, T5, T6
T3, T4, T5, T6
```

For each one of these combinations, the optimizer looks at all the join orders (permutations). For each set of 4 tables, there are 24 possible join orders, for a total of 360 (24 * 15) permutations. For example, for the set of tables T2, T3, T5, and T6, the optimizer looks at these 24 possible orders:

```
T2, T3, T5, T6
T2, T3, T6, T5
T2, T5, T3, T6
T2, T5, T6, T3
T2, T6, T3, T5
```

```
T2, T6, T5, T3
T3, T2, T5, T6
T3, T2, T6, T5
T3, T5, T2, T6
T3, T5, T6, T2
T3, T6, T2, T5
T3, T6, T5, T2
T5, T2, T3, T6
T5, T2, T6, T3
T5, T3, T2, T6
T5, T3, T6, T2
T5, T6, T2, T3
T5, T6, T3, T2
T6, T2, T3, T5
T6, T2, T5, T3
T6, T3, T2, T5
T6, T3, T5, T2
T6, T5, T2, T3
T6, T5, T3, T2
```

Let's say that the best join order is determined to be:

```
T5, T3, T6, T2
```

At this point, T5 is designated as the outermost table in the query.

The next step is to choose the second-outermost table. The optimizer eliminates T5 from consideration as it chooses the rest of the join order. Now, it has to determine where T1, T2, T3, T4, and T6 fit into the rest of the join order. It looks at all the combinations of four tables chosen from these five:

```
T1, T2, T3, T4
T1, T2, T3, T6
T1, T2, T4, T6
T1, T3, T4, T6
T2, T3, T4, T6
```

It looks at all the join orders for each of these combinations, remembering that T5 is the outermost table in the join. Let's say that the best order in which to join the remaining tables to T5 is:

```
T3, T6, T2, T4
```

So the optimizer chooses T3 as the next table after T5 in the join order for the entire query. It eliminates T3 from consideration in choosing the rest of the join order.

The remaining tables are:

```
T1, T2, T4, T6
```

Now we're down to 4 tables, so the optimizer looks at all the join orders for all the remaining tables. Let's say the best join order is:

```
T6, T2, T4, T1
```

This means that the join order for the entire query is:

```
T5, T3, T6, T2, T4, T1
```

### Outer joins and join permutations

Outer joins restrict the set of possible join orders. When the inner member of an outer join is compared to an outer member, the outer member must precede the inner member in the join order. The only join permutations that are considered for outer joins are those that meet this requirement. For example, these two queries perform outer joins, the first using ANSI SQL syntax, the second using Transact-SQL syntax:

```
select T1.c1, T2.c1, T3.c2, T4.c2
from T4 inner join T1 on T1.c1 = T4.c1
left outer join T2 on T1.c1 = T2.c1
left outer join T3 on T2.c2 = T3.c2
select T1.c1, T2.c1, T3.c2, T4.c2
from T1 , T2, T3, T4
where T1.c1 *= T2.c1
and T2.c2 *= T3.c2
and T1.c1 = T4.c1
```

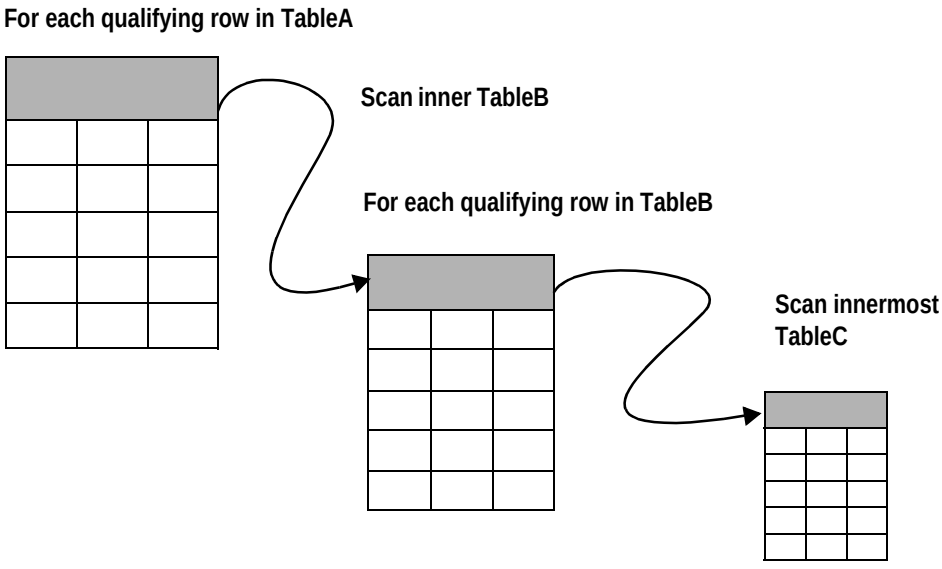
The only join orders considered place `T1` outer to `T2` and `T2` outer to `T3`. The join orders considered by the optimizer are:

```
T1, T2, T3, T4
T1, T2, T4, T3
T1, T4, T2, T3
T4, T1, T2, T3
```

## Nested-loop joins

Nested-loop joins provide efficient access when tables are indexed on join columns. The process of creating the result set for a nested-loop join is to nest the tables, and to scan the inner tables repeatedly for each qualifying row in the outer table, as shown in Figure 21-1.

***Figure 21-1: Nesting of tables during a nested-loop join***




In Figure 21-1, the access to the tables to be joined is nested:

- TableA is accessed once. If the table has no useful indexes, a table scan is performed. If an index can reduce I/O costs, the index is used to locate the rows.
- TableB is accessed once for each qualifying row in TableA. If 15 rows from TableA match the conditions in the query, TableB is accessed 15 times. If TableB has a useful index on the join column, it might require 3 I/Os to read the data page for each scan, plus one I/O for each data page. The cost of accessing TableB would be 60 logical I/Os.
- TableC is accessed once for each qualifying row in TableB *each time* TableB is accessed. If 10 rows from TableB match for each row in TableA, then TableC is scanned 150 times. If each access to TableC requires 3 I/Os to locate the data row, the cost of accessing TableC is 450 logical I/Os.

If TableC is small, or has a useful index, the I/O count stays reasonably small. If TableC is large and has no useful index on the join columns, the optimizer may choose to use a sort-merge join or the reformatting strategy to avoid performing extensive I/O.

## Cost formula

For a nested-loop join with two tables, the formula for estimating the cost is:

```
Join cost = Cost of accessing A +
                # of qualifying rows in A * Pages of B to scan for each qualifying row
```

With additional tables, the cost of a nested-loop join is:

```
      Cost of accessing outer table
+     (Number of qualified rows in outer) * ( Cost of accessing inner table)
+     ...
+     (Number of qualified rows from previous) * (Cost of accessing innermost table)
```

## How inner and outer tables are determined

The outer table is usually the one that has:

- The smallest number of qualifying rows, and/or
- The largest numbers of I/Os required to locate rows.

The inner table usually has:

- The largest number of qualifying rows, and/or
- The smallest number of reads required to locate rows.

For example, when you join a large, unindexed table to a smaller table with indexes on the join key, the optimizer chooses:

- The large table as the outer table, so that the large table is scanned only once.
- The indexed table as the inner table, so that each time the inner table is accessed, it takes only a few reads to find rows.

# Access methods and costing for sort-merge joins

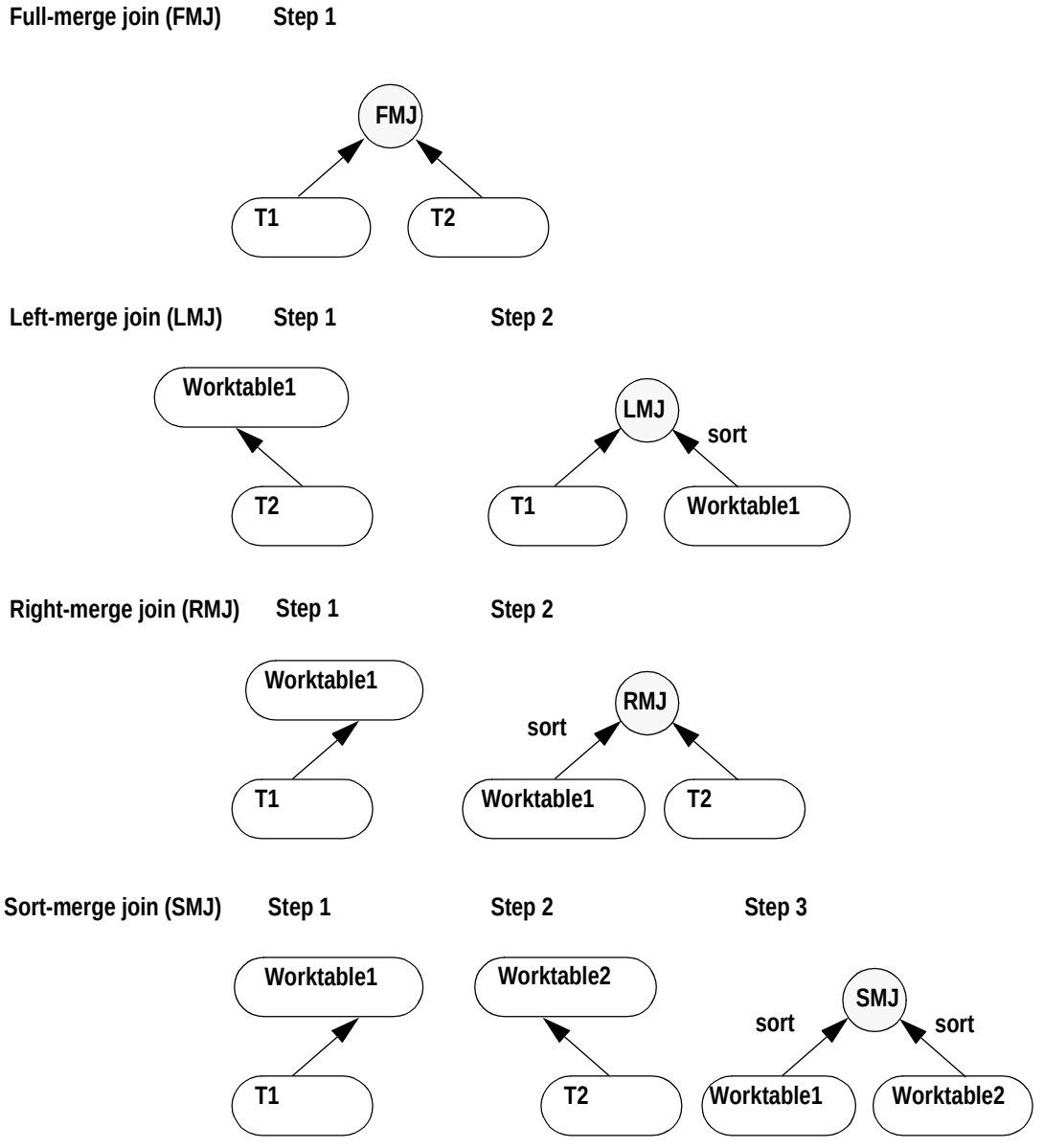
There are four possible execution methods for merge joins:

- Full-merge join – the two tables being joined have useful indexes on the join columns. The tables do not need to be sorted, but can be merged using the indexes.
- Left-merge join – sort the inner table in the join order, then merge with the left, outer table.
- Right-merge join – sort the outer table in the join order, then merge with the right, inner table.
- Sort-merge join – sort both tables, then merge.

Merge joins always operate on stored tables – either user tables or worktables created for the merge join. When a worktable is required for a merge join, it is sorted into order on the join key, then the merge step is performed. The costing for any merge joins that involve sorting includes the estimated I/O cost of creating and sorting a worktable. For full-merge joins, the only cost involved is scanning the tables.

Figure 21-2 provides diagrams of the merge join types.

***Figure 21-2: Merge join types***



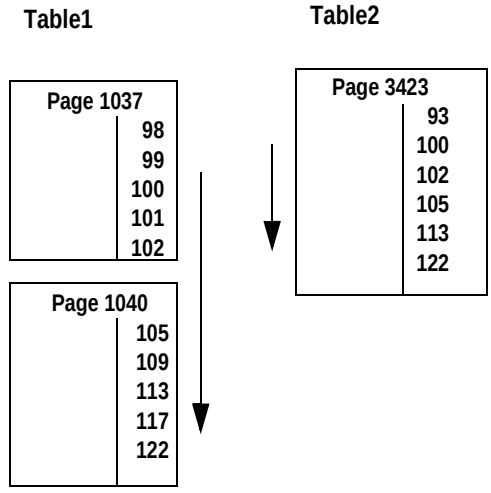
## How a full-merge is performed

If both Table1 and Table2 have indexes on the join key, this query can use a full-merge join:

```
select *
    from Table1, Table2
    where Table1.c1 = Table2.c2
    and Table1.c1 between 100 and 120
```

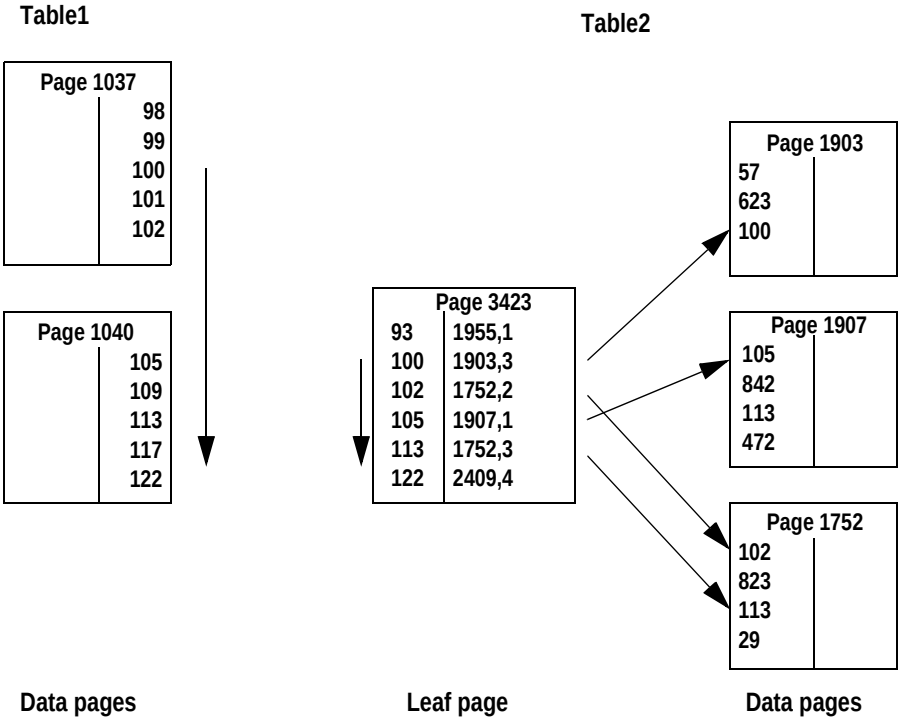
If both tables are allpages-locked tables with clustered indexes, and Table1 is chosen as the outer table, the index is used to position the search on the data page at the row where the value equals 100. The index on Table2 is also used to position the scan at the first row in Table2 where the join column equals 100. From this point, rows from both tables are returned as the scan moves forward on the data pages.

***Figure 21-3: A serial merge scan on two tables with clustered indexes***




Merge joins can also be performed using nonclustered indexes. The index is used to position the scan on the first matching value on the leaf page of the index. For each matching row, the index pointers are used to access the data pages. Figure 21-4 shows a full-merge scan using a nonclustered index on the inner table.

***Figure 21-4: Full merge scan using a nonclustered index on the inner table***




## How a right-merge or left-merge is performed

A right-merge or left-merge join always operates on a user table and a worktable created for the merge join. There are two steps:

1 A table or set of tables is scanned, and the results are inserted into a worktable.

2 The worktable is sorted and then merged with the other table in the join, using the index.

## How a sort-merge is performed

For a sort-merge join, there are three steps, since the inputs to the sort-merge joins are both sorted worktables:

1 A table or set of tables is scanned and the results are inserted into one worktable. This will be the outer table in the merge.

2 Another table is scanned and the results are inserted into another worktable. This will be the inner table in the merge.

3 Each of the worktables is sorted, then the two sorted result sets are merged.

## Mixed example

This query performs a mixture of merge and nested-loop joins:

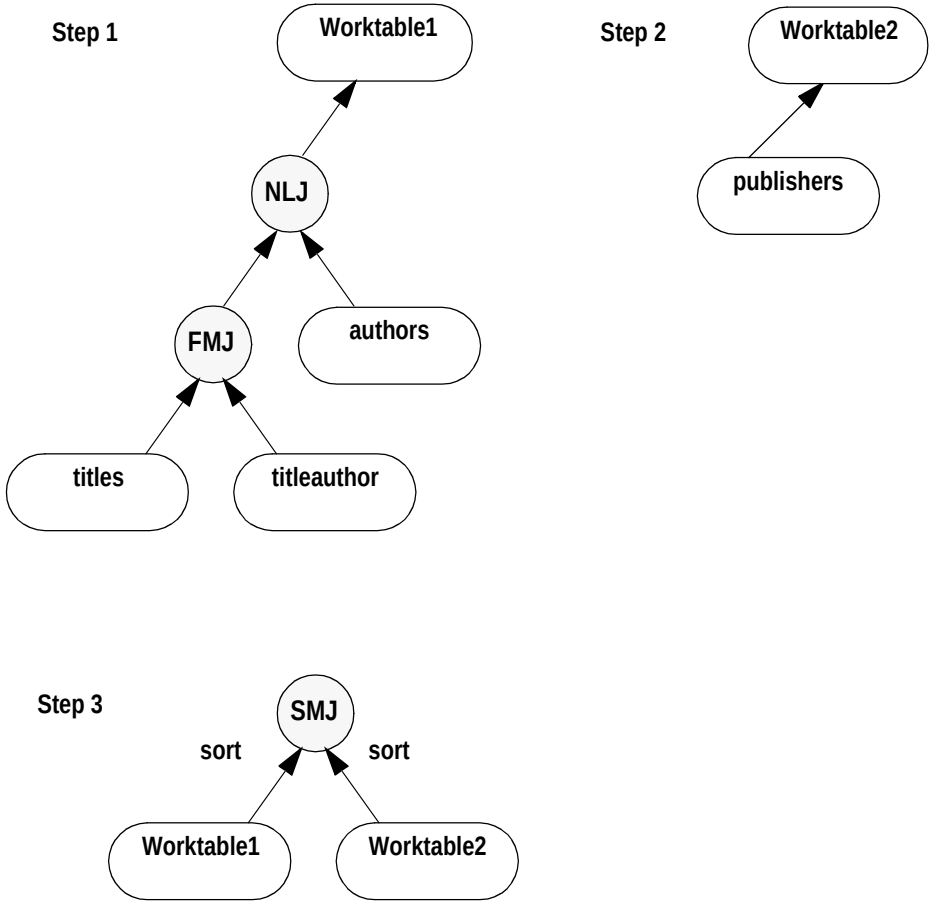
```
select pub_name, au_lname, price
from titles t, authors a, titleauthor ta,
            publishers p
where t.title_id = ta.title_id
    and a.au_id = ta.au_id
    and p.pub_id = t.pub_id
    and type = 'business'
    and price < $25
```

Adaptive Server executes this query in three steps:

- Step 1 uses 3 worker processes to scan titles as the outer table, performing a full-merge join with titleauthor and then a nested-loop join with authors. No sorting is required for the full-merge join. titles has a clustered index on title_id. The index on titleauthor, ta_ix, contains the title_id and au_id, so the index covers the query. The results are stored in Worktable1, for use in the sort-merge join performed in Step 3.
- Step 2 scans the publishers table, and saves the needed columns (pub_name and pub_id) in Worktable2.
- In Step 3:
  - Worktable1 is sorted into join column order, on pub_id.
  - Worktable2 is sorted into order on pub_id.
  - The sorted results are merged.

Figure 21-5 shows the steps.

***Figure 21-5: Multiple steps in processing a merge join***




## *showplan* messages for sort-merge joins

showplan messages for each type of merge join appear as specific combinations:

- Full-merge join – there are no “FROM TABLE Worktable” messages, only the “inner table” and “outer table” messages for base tables in the query.
- Right-merge join – the “outer table” is always a worktable.

- Left-merge join – the “inner table” is always a worktable.
- Sort-merge join – both tables are worktables.

For more information, see “Messages describing access methods, caching, and I/O cost” on page 793.

## Costing for merge joins

The total cost for merge joins depends on:

- The type of merge join.
  - Full-merge joins do not require sorts and worktables.
  - For right-merge and left-merge joins, one side of the join is selected into a worktable, then sorted.
  - For sort-merge joins, both sides of the join are selected into worktables, and each worktable is sorted.
- The type of index used to scan the tables while performing the merge step.
- The locking scheme of the underlying table: costing models for most scans are different for allpages locking than data-only locking. Clustered index access cost on data-only-locked tables is more comparable to nonclustered access.
- Whether the query is executed in serial or parallel mode.
- Whether the outer table has duplicate values for the join key.

In general, when comparing costs between a nested-loop join and a merge join for the same tables, using the same indexes, the cost for the outer table remains the same. Access to the inner table costs less for a merge join because the scan remains positioned on the leaf pages as matching values are returned, saving the logical I/O cost of scanning down the index from the root page each time.

## Costing for a full-merge with unique values

If a full-merge join is performed in serial mode and there is no need to sort the tables, the cost of a merge join on T1 and T2 is the sum of the cost of the scans of both tables, as long as all join values are unique:

Merge join cost = Cost of scan of T1 + Cost of scan of T2

The cost saving of a merge join over a nested-loop join is:

- For a nested-loop join, access to the inner table of the join starts at the root page of the index for each row from the outer table that qualifies.
- For a full-merge join, the upper levels of the index are used for the first access, to position the scan:
  - On the leaf page of the index, for nonclustered indexes and clustered indexes on data-only-locked tables
  - On the data page, if there is a clustered index on an allpages-locked table

  The higher levels of the index do not need to be read for each matching outer row.

## Example: allpages-locked tables with clustered indexes

For allpages-locked tables where clustered indexes are used to perform the scans, the search arguments on the index are used to position the search on the first matching row of each table. The total cost of the query is the cost of scanning forward on the data pages of each table. For example, with clustered indexes on t1(c1) and t2(c1), the query on two allpages-locked tables can use a full-merge join:

```
select t1.c2, t2.c2
from t1, t2
where t1.c1 = t2.c1
and t1.c1 >= 1000 and t1.c1 < 1100
```

If there are 100 rows that qualify from t1, and 100 rows from t2, and each of these tables has 10 rows per page, and an index height of 3, the costs are:

- 3 index pages to position the scan on the first matching row of t1
- Scanning 10 pages of t1

- 3 index pages to position the scan on the first matching row of t2
- Scanning 10 pages of t2

## Costing for a full-merge with duplicate values

If the outer table in a merge join has duplicate values, the inner table must be accessed from the root page of the index for each duplicate value. This query is the same as the previous example:

```
select t1.c2, t2.c2
from t1, t2
where t1.c1 = t2.c1
and t1.c1 >= 1000 and t1.c1 < 1100
```

If t1 is the outer table, and there are duplicate values for some of the rows in t1, so that there are 120 rows between 1000 and 1100,with 20 duplicate values, then each time one of the duplicate values is accessed, the scan of t2 is restarted from the root page of the index. If one row for t2 matches each value from t1, the I/O costs for this query are:

- 3 index pages to position on the first matching row of t1
- Scanning 12 pages of t1
- 3 index pages to position on the first matching row of t2, plus an I/O to read the data page
- For the remaining rows:
  - If the value from t1 is a duplicate, the scan of t2 restarts from the root page of the index.
  - For all values of t1 that are not duplicates, the scan remains positioned on the leaf level of t2. The scan on the inner table remains positioned on the leaf page as rows are returned until the next duplicate value in the outer table requires the scan to restart from the root page.

This formula gives the cost of the scan of the inner table for a merge join:

$$\text{Cost of scan of inner} = \text{Num duplicate values} * (\text{index height} + \text{scan size}) + \text{Num unique values} * \text{scan size}$$

The *scan size* is the number of pages of the inner table that need to be read for each value in the outer table. For tables where multiple inner rows match, the scan size is the average number of pages that need to be read for each outer row.

## Costing sorts

Sort cost during sort-merge joins depends on:

- The size of the worktables, which depends on the number of columns and rows selected
- The setting for the number of sort buffers configuration parameter, which determines how many pages of the cache can be used

These variables affect the number of merge runs required to sort the worktable.

### Worktable size for sort-merge joins

When a worktable is created for a merge join that requires a sort, only the columns that are needed for the result set and for later joins in the query execution are selected into the worktable. When the worktable for the titles table is created for the join shown in Figure 21-5 on page 484:

- Worktable1 includes the price and authors.state, because they are part of the result set, and pub_id, because it is needed for a subsequent join.
- Worktable2 includes the publishers.state column because it is part of the result set, and the pub_id, because it is needed for the merge step.

The type column is used as a search argument while the rows from titles are selected, but since it is not used later in the query or in the result set, it is not included in the worktable.

Each sort performed for a merge join can use up to number of sort buffers for intermediate sort steps. Sort buffers for worktable sorts are allocated from the cache used by tempdb. If the number of pages to be sorted is less the number of sort buffers, then the number of buffers reserved for the sort is the number of pages in the worktable.

## When merge joins cannot be used

Merge joins are not used:

- For joins using <, >, <=, >=, or != on the join columns.
- For outer joins, that is, queries using *= or =*, and left join and right join.
- For queries that include a text or image column or Java object columns in the select list or in a where clause.
- For subqueries that are not flattened or materialized in parallel queries.
- For multitable updates and deletes, such as:

```
update R set a = 5
    from R, S, T
    where ...
```

- For joins to perform referential integrity checks for insert, update, and delete commands. These joins are generated internally to check for the existence of the column values. They usually involve joins that return a single value from the referenced table. Often, these joins are supported by indexes. There would be no benefit from using a merge join for constraint checks.
- When the number of bytes in a row for a worktable would exceed the page-size limit (1960 bytes of user data) or the limit on the number of columns (1024). If the select list and required join columns for a join would create a worktable that exceeds either of these limits, the optimizer does not consider performing a merge join at that point in the query plan.
- When the use of worktables for a merge join would require more than the maximum allowable number of worktables for a query (14).

There are some limits on where merge joins can be used in the join order:

- Merge joins can be performed only before an existence join. Some distinct queries are turned into existence joins, and merge joins are not used for these.
- Full-merge joins and left-merge joins can be performed only on the outermost tables in the join order.

## Use of worker processes

When parallel processing is enabled, merge joins can use multiple worker processes to perform:

- The scan that selects rows into the worktables
- Worktable sort operations
- The merge join and subsequent joins in the step

See “Parallel range-based scans” on page 546 for more information.

## Recommendations for improved merge performance

Here are some suggestions for improving sort-merge join performance:

- To reduce the size of worktables select only needed columns for tables used in merge joins. Avoid using select * unless you need all columns of the tables. This reduces the load on tempdb and the cost of sorting the result tables.
- If you are concerned about possible performance impacts of merge joins or possible space problems in tempdb, see Chapter 28, “Introduction to Abstract Plans,” for a discussion of how abstract query plans can help determine which queries on your system use merge joins.
- Look for opportunities for index covering. One example is queries where joins are in the form:

  ```
  select t1.c3, t3.c4
  from t1, t2, t3
  wehre t1.c1 = t2.c1 and t2.c2 = t3.c2
  and ...
  ```

  and columns from *t2* are not in the select list, or only the join columns are in the select list. An index on the join columns, *t2(c1, c2)* covers the query, allowing a merge join to avoid accessing the data pages of t2.
- Merge joins can use indexes created in ascending or descending order when two tables are joined on multiple columns, such as these:

  ```
  A.c1 = B.c1 and A.c2 = B.c2 and A.c3 = B.c3
  ```

The column order specified for the indexes must be an exact match, or exactly the reverse, for all columns to be used as join predicates when costing the join and accessing the data. If there is a mismatch of ordering in second or subsequent columns, only the matching columns are used for the join, and the remaining columns are used to restrict the results after the row has been retrieved. This table shows some examples for the query above:

| Index creation order | Clauses used as join predicates |
|---|---|
| A(c1 asc, c2 asc, c3 asc)<br>B(c1 asc, c2 asc, c3 asc) | All three clauses. |
| A(c1 asc, c2 asc, c3 asc)<br>B(c1 desc, c2 desc, c3 desc) | All three clauses. |
| A(c1 asc, c2 asc, c3 asc)<br>B(c1 desc, c2 desc, c3 asc) | The first two join clauses are used as join predicates and the third clause is evaluated as a restriction on the result. |
| A1(c1 asc, c2 desc, c3 desc)<br>B1(c1 desc, c2 desc, c3 asc) | Only the first join clause is used as a join predicate. The remaining two clauses is evaluated as restrictions on the result set. |

Index key ordering is generally chosen to eliminate sort costs for order by queries. Using compatible ordering for frequently joined tables can also reduce join costs.

# Enabling and disabling merge joins

You can enable and disable merge joins at the server and session level using set sort_merge, or at the server level with the configuration parameter enable sort-merge joins and JTC. This configuration parameter also enables and disables join transitive closure.

## At the server level

To enable merge joins server-wide, set enable sort-merge joins and JTC to 1. The default value is 0, which means that merge joins are not considered. When this value is set to 1, merge joins and join transitive closure are considered for equijoins. If merge joins are disabled at the server level, they can be enabled for a session with set sort_merge.

Join transitive closure can be enabled independently at the session level with set jtc on.

See “Enabling and disabling join transitive closure” on page 418.

The configuration parameter is dynamic, and can be reset without restarting the server.

## At the session level

To enable merge joins for a session, use:

```
set sort_merge on
```

To disable merge joins during a session, use:

```
set sort_merge off
```

The session setting has precedence over the server-wide setting; you can use merge joins in a session or stored procedure even if they are disabled at the server-wide level.

# Reformatting strategy

When a table is large and has no useful index for a join, the optimizer considers a sort merge join, and also considers creating and sorting a worktable, and using a nested-loop join.

The process of generating a worktable with a clustered index and performing a nested-loop join is known as ***reformatting***.

Like a sort-merge join, reformatting scans the tables and copies qualifying rows to a worktable. But instead of the sort and merge used for a merge join, Adaptive Server creates a temporary clustered index on the join column for the inner table. In some cases, creating and using the clustered index is cheaper than a sort-merge join.

The steps in the reformatting strategy are:

- Creating a worktable
- Inserting the needed columns from the qualifying rows
- Creating a clustered index on the join columns of the worktable
- Using the clustered index in the join to retrieve the qualifying rows from each table

The main cost of the reformatting strategy is the time and I/O necessary to create the worktable and to build the clustered index on the worktable. Adaptive Server uses reformatting only when the reformatting cost is less than the cost of a merge join or repeated table scans.

A showplan message indicates when Adaptive Server is using the reformatting strategy and includes other messages showing the steps used to build the worktables.

See “Reformatting Message” on page 809.

# Subquery optimization

Subqueries use the following optimizations to improve performance:

- Flattening – converting the subquery to a join
- Materializing – storing the subquery results in a worktable
- Short circuiting – placing the subquery last in the execution order
- Caching subquery results – recording the results of executions

The following sections explain these strategies.

See “showplan messages for subqueries” on page 819 for an explanation of the showplan messages for subquery processing.

## Flattening *in*, *any*, and *exists* subqueries

Adaptive Server can flatten some quantified predicate subqueries to a join. Quantified predicate subqueries are introduced with in, any, or exists. Each result row in the outer query is returned once, and only once, if the subquery condition evaluates to TRUE.

### When flattening can be done

- For any level of nesting of subqueries, for example:

```
select au_lname, au_fname
from authors
where au_id in
   (select au_id
    from titleauthor
    where title_id in
       (select title_id
        from titles
        where type = "popular_comp") )
```

- For multiple subqueries in the outer query, for example:

```
select title, type
from titles
where title in
   (select title
    from titles, titleauthor, authors
   where titles.title_id = titleauthor.title_id
    and titleauthor.au_id = authors.au_id
    and authors.state = "CA")
and title in
   (select title
    from titles, publishers
    where titles.pub_id = publishers.pub_id
    and publishers.state = "CA")
```

### Exceptions to flattening

A subquery introduced with in, any, or exists cannot be flattened if one of the following is true:

- The subquery is correlated and contains one or more aggregates.
- The subquery is in the select list or in the set clause of an update statement.

- The subquery is connected to the outer query with or.
- The subquery is part of an isnull predicate.
- The subquery is the outermost subquery in a case expression.

If the subquery computes a scalar aggregate, materialization rather than flattening is used.

See "Materializing subquery results" on page 499.

## Flattening methods

Adaptive Server uses one of these flattening methods to resolve a quantified predicate subquery using a join:

- A regular join – if the uniqueness conditions in the subquery mean that it returns a unique set of values, the subquery can be flattened to use a regular join.
- An existence join, also known as a semi-join – instead of scanning a table to return all matching values, an existence join returns TRUE when it finds the first matching value and then stops processing. If no matching value is found, it returns FALSE.
- A unique reformat – the subquery result set is selected into a worktable, sorted to remove duplicates, and a clustered index is built on the worktable. The clustered index is used to perform a regular join.
- A duplicate elimination sort optimization – the subquery is flattened into a regular join that selects the results into a worktable, then the worktable is sorted to remove duplicate rows

## Join order and flattening methods

A major factor in the choice of flattening method depends on the cost of the possible join orders. For example, in a join of t1, t2, and t3:

```
select * from t1, t2
where t1.c1 = t2.c1
and t2.c2 in (select c3 from t3)
```

If the cheapest join order is t1, t2, t3 or t2, t1, t3, a regular joinor or an existence join is used. However, if it is cheaper to perform the join with t3 as the outer table, say, t3, t1, t2, a unique reformat or duplicate elimination sort is used.

The resulting flattened join can include nested-loop joins or merge joins. When an existence join is used, merge joins can be performed only before the existence join.

## Flattened subqueries executed as regular joins

Quantified predicate subqueries can be executed as normal joins when the result set of the subquery is a set of unique values. For example, if there is a unique index on publishers.pub_id, this single-table subquery is guaranteed to return a set of unique values:

```
select title
from titles
where pub_id in (select pub_id
    from publishers
    where state = "TX")
```

With a nonunique index on publishers.city, this query can also be executed using a regular join:

```
select au_lname
from authors a
where exists (select city
        from publishers p where p.city = a.city)
```

Although the index on publishers.city is not unique, the join can still be flattened to a normal join if the index is used to filter duplicate rows from the query.

When a subquery is flattened to a normal join, showplan output shows a normal join. If filtering is used, showplan output is not different; the only diagnostic message is in dbcc traceon(310) output, where the *method* for the table indicates "NESTED ITERATION with Tuple Filtering."

## Flattened subqueries executed as existence joins

All in, any, and exists queries test for the existence of qualifying values and return TRUE as soon as a matching row is found.

The optimizer converts the following subquery to an existence join:

```
select title
    from titles
    where title_id in
        (select title_id
         from titleauthor)
    and title like "A Tutorial%"
```

The existence join query looks like the following ordinary join, although it does not return the same results:

```
select title
    from titles T, titleauthor TA
    where T.title_id = TA.title_id
        and title like "A Tutorial%"
```

In the pubtune database, two books match the search string on title. Each book has multiple authors, so it has multiple entries in titleauthor. A regular join returns five rows, but the subquery returns only two rows, one for each title_id, since it stops execution of the join at the first matching row.

When subqueries are flattened to use existence joins, the showplan output shows output for a join, with the message "EXISTS TABLE: nested iteration" as the join type for the table in the subquery.

## Flattened subqueries executed using unique reformatting

To perform unique reformatting, Adaptive Server:

- Selects rows into a worktable and sorts the worktable, removing duplicates and creating a clustered index on the join key.
- Joins the worktable with the next table in the join order. If there is a nonunique index on publishers.pub_id, this query can use a unique reformat strategy:

  ```
  select title_id
  from titles
  where pub_id in
  (select pub_id from publishers where state =
  "TX")
  ```

  This query is executed as:

  ```
  select pub_id
  into #publishers
  from publishers
  where state = "TX"
  ```

And after the sort removes duplicates and creates the clustered index:

```
select title_id
from titles, #publishers
where titles.pub_id = #publishers.pub_id
```

showplan messages for unique reformatting show “Worktable created for REFORMATTING” in Step 1, and “Using Clustered Index” on the worktable in Step 2.

dbcc traceon(310) displays “REFORMATTING with Unique Reformatting” for the method for the publishers table.

## Flattened subqueries using duplicate elimination

When it is cheaper to place the subquery tables as outer tables in the join order, the query is executed by:

- Performing a regular join with the subquery flattened into the outer query, placing results in a worktable.
- Sorting the worktable to remove duplicates.

For example, salesdetail has duplicate values for title_id, and it is used in this subquery:

```
select  title_id, au_id, au_ord
from titleauthor ta
where title_id in (select ta.title_id
    from titles t, salesdetail sd
    where t.title_id = sd.title_id
    and ta.title_id = t.title_id
    and type = 'travel' and qty > 10)
```

If the best join order for this query is salesdetail, titles, titleauthor, the optimal join order can be used by:

- Selecting all of the query results into a worktable
- Removing the duplicates from the worktable and returning the results to the user

showplan Messages for Flattened Subqueries Performing Sorts

showplan output includes two steps for subqueries that use normal joins plus a sort. The first step shows “Worktable1 created for DISTINCT” and the flattened join. The second step shows the sort and select from the worktable.

dbcc traceon(310) prints a message for each join permutation when a table or tables from a quantified predicate subquery is placed first in the join order. Here is the output when the join order used for the query above is considered:

```
2 - 0 - 1 -
```

```
This join order created while converting an exists
join to a regular join, which can happen for
subqueries, referential integrity, and select
distinct.
```

## Flattening expression subqueries

Expression subqueries are included in a query's select list or that are introduced by >, >=, <, <=, =, or !=. Adaptive Server converts, or flattens, expression subqueries to **equijoins** if:

- The subquery joins on unique columns or returns unique columns, and
- There is a unique index on the columns.

## Materializing subquery results

In some cases, a subquery is processed in two steps: the results from the inner query are *materialized*, or stored in a temporary worktable or internal variable, before the outer query is executed. The subquery is executed in one step, and the results of this execution are stored and then used in a second step. Adaptive Server materializes these types of subqueries:

- Noncorrelated expression subqueries
- Quantified predicate subqueries containing aggregates where the having clause includes the correlation condition

### Noncorrelated expression subqueries

Noncorrelated expression subqueries must return a single value. When a subquery is not correlated, it returns the same value, regardless of the row being processed in the outer query. The query is executed by:

- Executing the subquery and storing the result in an internal variable.
- Substituting the result value for the subquery in the outer query.

The following query contains a noncorrelated expression subquery:

```
select title_id
from titles
where total_sales = (select max(total_sales)
```

```
                        from ts_temp)
```

Adaptive Server transforms the query to:

```
select <internal_variable> = max(total_sales)
    from ts_temp
select title_id
    from titles
    where total_sales = <internal_variable>
```

The search clause in the second step of this transformation can be optimized. If there is an index on total_sales, the query can use it. The total cost of a materialized expression subquery is the sum of the cost of the two separate queries.

## Quantified predicate subqueries containing aggregates

Some subqueries that contain vector (grouped) aggregates can be materialized. These are:

- Noncorrelated quantified predicate subqueries
- Correlated quantified predicate subqueries correlated only in the having clause

The materialization of the subquery results in these two steps:

- Adaptive Server executes the subquery first and stores the results in a worktable.
- Adaptive Server joins the outer table to the worktable as an existence join. In most cases, this join cannot be optimized because statistics for the worktable are not available.

Materialization saves the cost of evaluating the aggregates once for each row in the table. For example, this query:

```
select title_id
from titles
where total_sales in (select max(total_sales)
                      from titles
                      group by type)
```

Executes in these steps:

```
select maxsales = max(total_sales)
    into #work
    from titles
    group by type
select title_id
```

```
from titles, #work
where total_sales = maxsales
```

The total cost of executing quantified predicate subqueries is the sum of the query costs for the two steps.

When there are where clauses in addition to a subquery, Adaptive Server executes the subquery or subqueries last to avoid unnecessary executions of the subqueries. Depending on the clauses in the query, it is often possible to avoid executing the subquery because less expensive clauses can determine whether the row is to be returned:

- If any and clauses evaluate to FALSE, the row will not be returned.
- If any or clauses evaluate to TRUE, the row will be returned.

In both cases, as soon as the status of the row is determined by the evaluation of one clause, no other clauses need to be applied to that row. This provides a performance improvement, because expensive subqueries need to be executed less often.

## Subquery introduced with an *and* clause

When and joins the clauses, evaluation stops as soon as any clause evaluates to FALSE. The row is skipped.

This query contains two and clauses, in addition to the correlated subquery:

```
select au_fname, au_lname, title, royaltyper
from titles t, authors a, titleauthor ta
where t.title_id = ta.title_id
and a.au_id = ta.au_id
and advance >= (select avg(advance)
                        from titles t2
                        where t2.type = t.type)
and price > $100
and au_ord = 1
```

Adaptive Server orders the execution steps to evaluate the subquery last, after it evaluates the conditions on price and au_ord. If a row does not meet an and condition, Adaptive Server discards the row without checking any more and conditions and begins to evaluate the next row, so the subquery is not processed unless the row meets all of the and conditions.

## Subquery introduced with an *or* clause

If a query's where conditions are connected by or, evaluation stops when any clause evaluates to TRUE, and the row is returned.

This query contains two or clauses in addition to the subquery:

```
select au_fname, au_lname, title
from titles t, authors a, titleauthor ta
where t.title_id = ta.title_id
and a.au_id = ta.au_id
and (advance > (select avg(advance)
                    from titles t2
                    where t.type = t2.type)
or title = "Best laid plans"
or price > $100)
```

Adaptive Server orders the conditions in the query plan to evaluate the subquery last. If a row meets the condition of the or clause, Adaptive Server returns the row without executing the subquery, and proceeds to evaluate the next row.

## Subquery results caching

When it cannot flatten or materialize a subquery, Adaptive Server uses an in-memory cache to store the results of each evaluation of the subquery. While the query runs, Adaptive Server tracks the number of times a needed subquery result is found in cache. This is called a **cache hit ratio**. If the cache hit ratio is high, it means that the cache is reducing the number of times that the subquery executes. If the cache hit ratio is low, the cache is not useful, and it is reduced in size as the query runs.

Caching the subquery results improves performance when there are duplicate values in the join columns or the correlation columns. It is even more effective when the values are ordered, as in a query that uses an index. Caching does not help performance when there are no duplicate correlation values.

### Displaying subquery cache information

The set statistics subquerycache on command displays the number of cache hits and misses and the number of rows in the cache for each subquery. The following example shows subquery cache statistics:

```
set statistics subquerycache on
```

```
select type, title_id
from titles
where price > all
    (select price
        from titles
        where advance < 15000)
Statement: 1  Subquery: 1  cache size: 75  hits: 4925
misses: 75
```

If the statement includes subqueries on either side of a union, the subqueries are numbered sequentially through both sides of the union.

## Optimizing subqueries

When queries containing subqueries are not flattened or materialized:

- The outer query and each unflattened subquery are optimized one at a time.
- The innermost subqueries (the most deeply nested) are optimized first.
- The estimated buffer cache usage for each subquery is propagated outward to help evaluate the I/O cost and strategy of the outer queries.

In many queries that contain subqueries, a subquery is “nested over” to one of the outer table scans by a two-step process. First, the optimizer finds the point in the join order where all the correlation columns are available. Then, the optimizer searches from that point to find the table access that qualifies the fewest rows and attaches the subquery to that table. The subquery is then executed for each qualifying row from the table it is nested over.

# *or* Clauses versus unions in joins

Adaptive Server cannot optimize join clauses that are linked with or and it may perform Cartesian products to process the query.

**Note** Adaptive Server optimizes search arguments that are linked with or. This description applies only to join clauses.

For example, when Adaptive Server processes this query, it must look at every row in one of the tables for each row in the other table:

```
select *
    from tab1, tab2
    where tab1.a = tab2.b
        or tab1.x = tab2.y
```

If you use union, each side of the union is optimized separately:

```
    select *
        from tab1, tab2
        where tab1.a = tab2.b
union all
    select *
        from tab1, tab2
        where tab1.x = tab2.y
```

You can use union instead of union all to eliminate duplicates, but this eliminates all duplicates. You may not get exactly the same set of duplicates from the rewritten query.

Adaptive Server can optimize selects with joins that are linked with union. The result of or is somewhat like the result of union, except for the treatment of duplicate rows and empty tables:

- union removes all duplicate rows (in a sort step); union all does not remove any duplicates. The comparable query using or might return some duplicates.
- A join with an empty table returns no rows.

CHAPTER 22

# Parallel Query Processing

This chapter introduces basic concepts and terminology needed for parallel query optimization, parallel sorting, and other parallel query topics, and provides an overview of the commands for working with parallel queries.



Other chapters that cover specific parallel processing topics in more depth include:

- For details on how the Adaptive Server optimizer determines eligibility and costing for parallel execution, see Chapter 23, "Parallel Query Optimization."
- To understand parallel sorting topics, see Chapter 24, "Parallel Sorting."
- For information on object placement for parallel performance, see "Partitioning tables for performance" on page 85.
- For information about locking behavior during parallel query processing, see *System Administration Guide*
- For information on showplan messages, see "showplan messages for parallel queries" on page 814.
- To understand how Adaptive Server uses multiple engines, see Chapter 3, "Using Engines and CPUs."

# Types of queries that can benefit from parallel processing

When Adaptive Server is configured for parallel query processing, the optimizer evaluates each query to determine whether it is eligible for parallel execution. If it is eligible, and if the optimizer determines that a parallel query plan can deliver results faster than a serial plan, the query is divided into components that are processed simultaneously. The results are combined and delivered to the client in a shorter period of time than it would take to process the query serially as a single component.

Parallel query processing can improve the performance of the following types of queries:

- select statements that scan large numbers of pages but return relatively few rows, such as:
  - Table scans or clustered index scans with grouped or ungrouped aggregates
  - Table scans or clustered index scans that scan a large number of pages, but have where clauses that return only a small percentage of the rows
- select statements that include union, order by, or distinct, since these queries can populate worktables in parallel, and can make use of parallel sorting
- select statements that use merge joins can use parallel processing for scanning tables and for performing the sort and merge steps
- select statements where the reformatting strategy is chosen by the optimizer, since these can populate worktables in parallel, and can make use of parallel sorting
- create index statements, and the alter table...add constraint clauses that create indexes, unique and primary key
- The dbcc checkstorage command

Join queries can use parallel processing on one or more tables.

Commands that return large, unsorted result sets are unlikely to benefit from parallel processing due to network constraints—in most cases, results can be returned from the database faster than they can be merged and returned to the client over the network.

Commands that modify data (insert, update, and delete), and cursors do not run in parallel. The inner, nested blocks of queries containing subqueries are never executed in parallel, but the outer block can be executed in parallel.

Decision support system (DSS) queries that access huge tables and return summary information benefit the most from parallel query processing. The overhead of allocating and managing parallel queries makes parallel execution less effective for online transaction processing (OLTP) queries, which generally access fewer rows and join fewer tables. When a server is configured for parallel processing, only queries that access 20 data pages or more are considered for parallel processing, so most OLTP queries run in serial.

## Adaptive Server's worker process model

Adaptive Server uses a **coordinating process** and multiple **worker processes** to execute queries in parallel. A query that runs in parallel with eight worker processes is much like eight serial queries accessing one-eighth of the table, with the coordinating process supervising the interaction and managing the process of returning results to the client. Each worker process uses approximately the same amount of memory as a user connection. Each worker process runs as a task that must be scheduled on an engine, scans data pages, queues disk I/Os, and performs in many ways like any other task on the server. One major difference is that in last phase of query processing, the coordinating process manages merging the results and returning them to the client, coordinating with worker processes.

Figure 22-1 shows the events that take place during parallel query processing:

1 The client submits a query.

2 The client task assigned to execute the query becomes the coordinating process for parallel query execution.

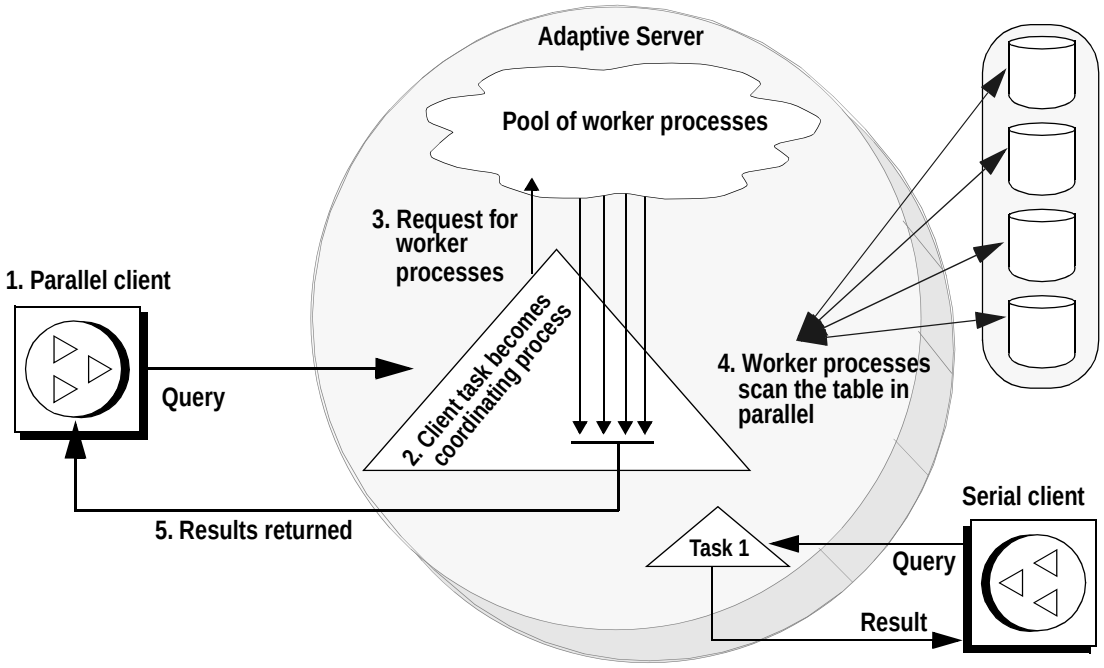
3 The coordinating process requests four worker processes from the pool of worker processes. The coordinating process together with the worker processes is called a **family**.

4 The worker processes execute the query in parallel.

5 The coordinating process returns the results produced by all the worker processes.

The serial client shown in the lower-right corner of Figure 22-1 submits a query that is processed serially.

***Figure 22-1: Worker process model***

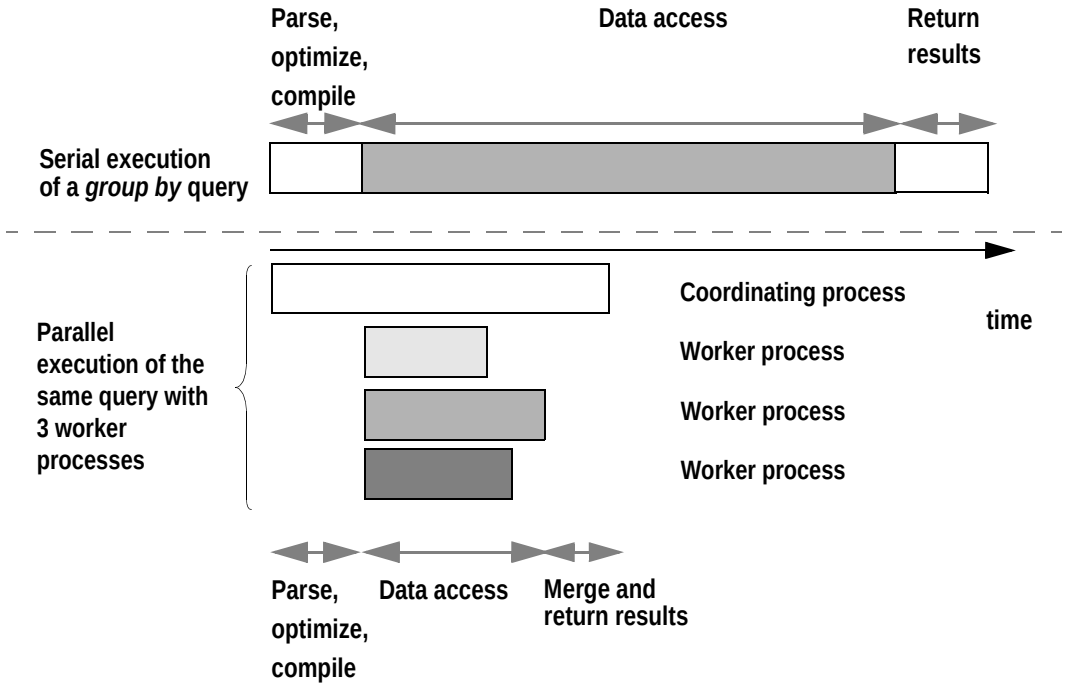



During query processing, the tasks are tracked in the system tables by a family ID (fid). Each worker process for a family has the same family ID and its own unique server process ID (spid). System procedures such as sp_who and sp_lock display both the fid and the spid for parallel queries, allowing you to observe the behavior of all processes in a family.

## Parallel query execution

Figure 22-2 shows how parallel query processing reduces response time over the same query running in serial. In parallel execution, three worker processes scan the data pages. The times required by each worker process may vary, depending on the amount of data that each process needs to access. Also, a scan can be temporarily blocked due to locks on data pages held by other users. When all of the data has been read, the results from each worker process are merged into a single result set by the coordinating process and returned to the client.

***Figure 22-2: Relative execution times for serial and parallel query execution***




The total amount of work performed by the query running in parallel is greater than the amount of work performed by the query running in serial, but the response time is shorter.

## Returning results from parallel queries

Results from parallel queries are returned through one of three merge strategies, or as the final step in a sort. Parallel queries that do not have a final sort step use one of these merge types:

- Queries that contain a vector (grouped) aggregate use worktables to store temporary results; the coordinating process merges the results into one worktable and returns results to the client.
- Queries that contain a scalar (ungrouped) aggregate use internal variables, and the coordinating process performs the final computations to return the results to the client.
- Queries that do not contain aggregates and that do not use clauses that do not require a final sort can return results to the client as the tables are being scanned. Each worker process stores results in a result buffer and uses address locks to coordinate transferring the results to the network buffers for the task.

More than one merge type can be used when queries require several steps or multiple worktables.

See "showplan messages for parallel queries" on page 814 for more information on merge messages.

For parallel queries that include an order by clause, distinct, or union, results are stored in a worktable in tempdb, then sorted. If the sort can benefit from parallel sorting, a parallel sort is used, and results are returned to the client during the final merge step performed by the sort.

For more information on how parallel sorts are performed, see Chapter 24, "Parallel Sorting."

---

**Note** Since parallel queries use multiple processes to scan data pages, queries that do not use aggregates and do not include a final sort step may return results in different order than serial queries and may return different results for queries with set rowcount in effect and for queries that select into a local variable.

For details and solutions, see "When parallel query results can differ" on page 533.

---

# Types of parallel data access

Adaptive Server accesses data in parallel in different ways, depending configuration parameter settings, table partitioning, and the availability of indexes. The optimizer may choose a mix of serial and parallel methods for queries that involve multiple tables or multiple steps. Parallel methods include:

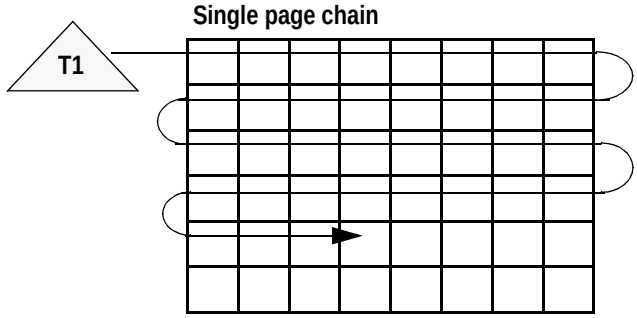
- Hash-based table scans
- Hash-based nonclustered index scans
- Partition-based scans, either full table scans or scans positioned with a clustered index
- Range-based scans during merge joins

The following sections describe some of the methods.

For more examples, see Chapter 23, “Parallel Query Optimization.”

Figure 22-3 shows a scan on an allpages-locked table executed in serial by a single task. The task follows the table’s page chain to read each page, stopping to perform physical I/O when needed pages are not in the cache.

***Figure 22-3: A serial task scans data pages***

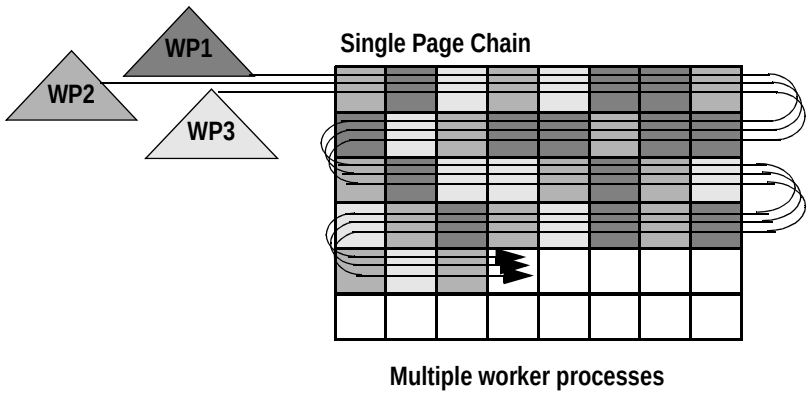


## Hash-based table scans

Figure 22-4 shows how three worker processes divide the work of accessing data pages from an allpages-locked table during a hash-based table scan. Each worker process performs a logical I/O on every page, but each process examines rows on only one-third of the pages, as indicated by the differently shaded pages. Hash-based table scans are used only for the outer query in a join.

With only one engine, the query still benefits from parallel access because one worker process can execute while others wait for I/O. If there are multiple engines, some of the worker processes could be running simultaneously.

***Figure 22-4: Worker processes scan an unpartitioned table***

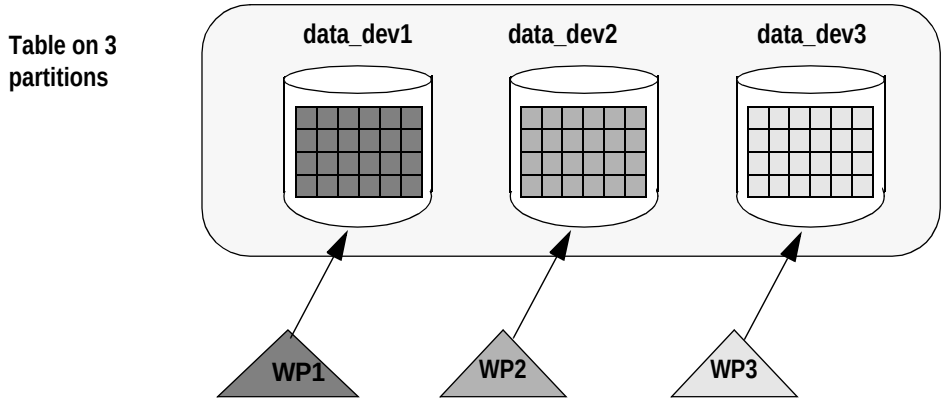



Hash-based table scans increase the logical I/O for the scan, since each worker process must access each page to hash on the page ID. For data-only-locked tables, hash-based table scans hash either on the extent ID or the allocation page ID, so that only a single worker process scans a page, and logical I/O does not increase.

## Partition-based scans

Figure 22-5 shows how a query scans a table that has three partitions on three physical disks. With a single engine, this query can benefit from parallel processing because one worker process can execute while others sleep waiting for I/O or waiting for locks held by other processes to be released. If multiple engines are available, the worker processes can run simultaneously. This configuration can yield high parallel performance by providing I/O parallelism.

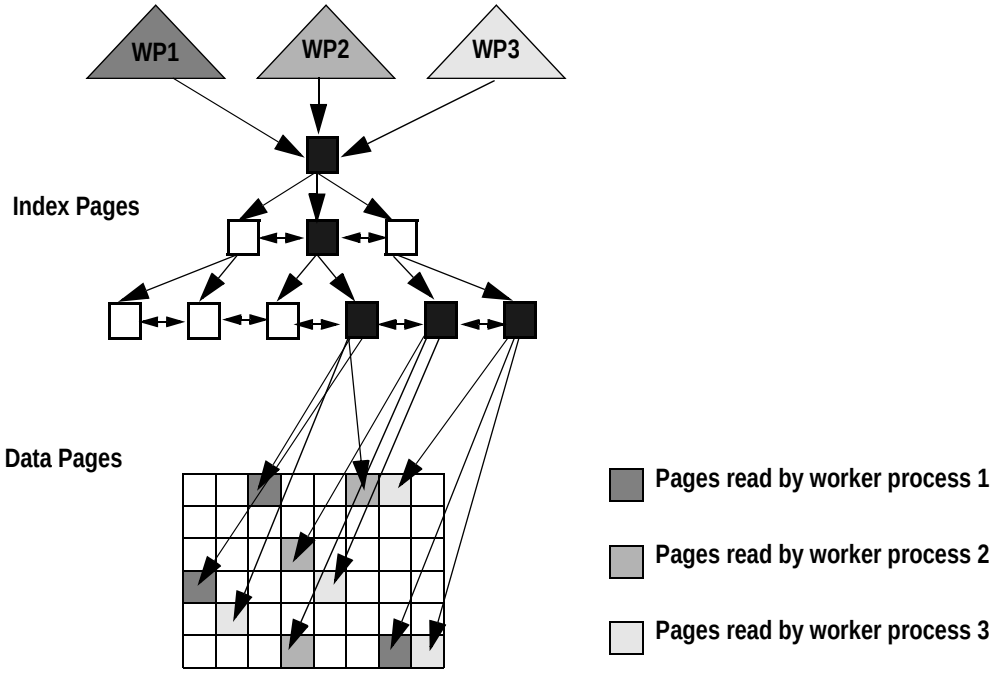
***Figure 22-5: Multiple worker processes access multiple partitions***




## Hash-based index scans

Figure 22-6 shows a hash-based index scan. Hash-based index scans can be performed using nonclustered indexes or clustered indexes on data-only-locked tables. Each worker process navigates higher levels of the index and reads the leaf-level pages of the index. Each worker process then hashes on either the data page ID or the key value to determine which data pages or data rows to process. Reading every leaf page produces negligible overhead.

***Figure 22-6: Hash-based, nonclustered index scan***

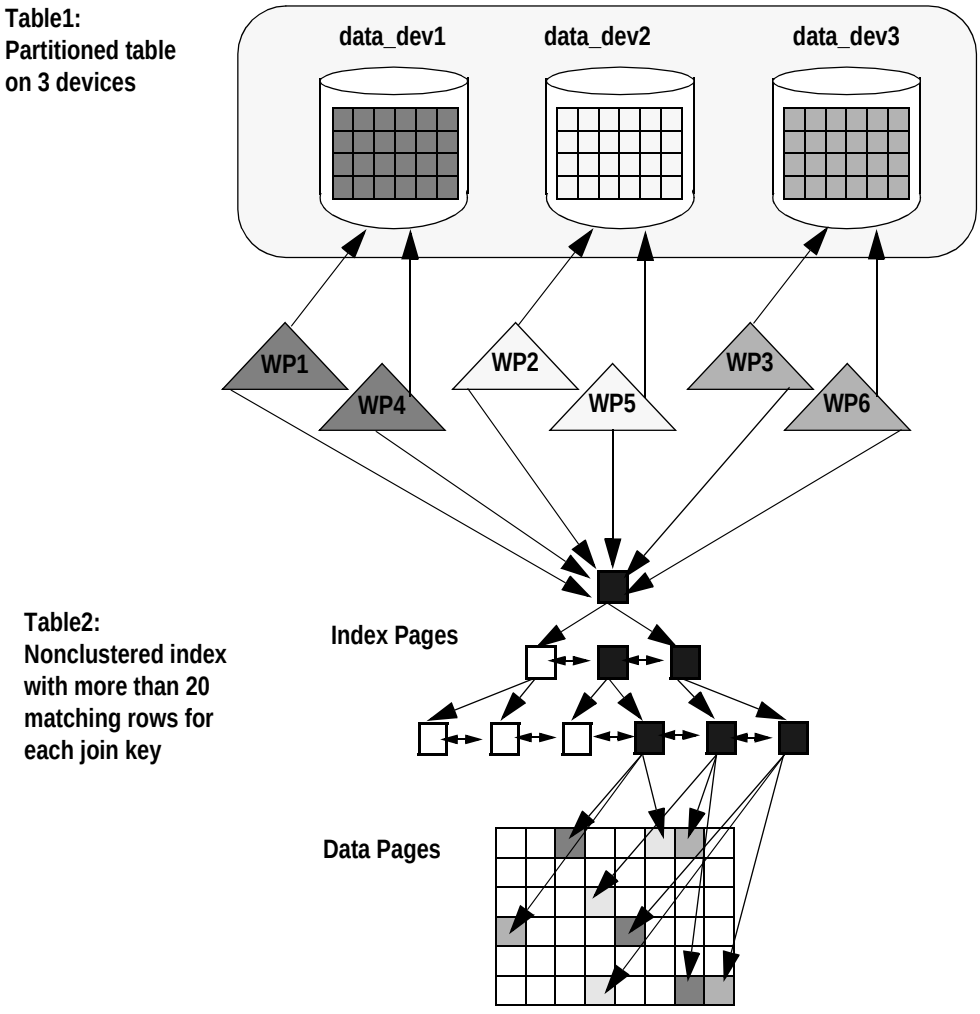



## Parallel processing for two tables in a join

Figure 22-7 shows a nested-loop join query performing a partition-based scan on a table with three partitions, and a hash-based index scan, with two worker processes on the second table. When parallel access methods are used on more than one table in a nested-loop join, the total number of worker processes required is the product of worker process for each scan. In this case, six workers perform the query, with each worker process scanning both tables. Two worker processes scan each partition in the first table, and all six worker processes navigate the index tree for the second table and scan the leaf pages. Each worker process accesses the data pages that correspond to its hash value.

The optimizer chooses a parallel plan for a table only when a scan returns 20 pages or more. These types of join queries require 20 or more matches on the join key for the inner table in order for the inner scan to be optimized in parallel.

***Figure 22-7: Join query using different parallel access methods on each table***




## *showplan* messages

showplan prints the degree of parallelism each time a table is accessed in parallel. The following example shows the messages for each table in the join in Figure 22-7:

```
Executed in parallel with a 2-way hash scan.
Executed in parallel with a 3-way partition scan.
```

showplan also prints a message showing the total number of worker processes used. For the query shown in Figure 22-7, it reports:

```
Executed in parallel by coordinating process and 6
worker processes.
```

See “showplan messages for parallel queries” on page 814 for more information and Chapter 23, “Parallel Query Optimization,” for additional examples.

# Controlling the degree of parallelism

A parallel query’s **degree of parallelism** is the number of worker processes used to execute the query. This number depends on several factors, including:

- The values to which of the parallel configuration parameters or the session-level limits,

  (see Table 22-1 and Table 22-2)
- The number of partitions on a table (for partition-based scans)
- The level of parallelism suggested by the optimizer
- The number of worker processes that are available at the time the query executes.

You can establish limits on the degree of parallelism:

- Server-wide – using sp_configure with parameters shown in Table 22-1. Only a System Administrator can use sp_configure.
- For a session – using set with the parameters shown in Table 22-2. All users can run set; it can also be included in stored procedures.
- In a select query – using the parallel clause, as shown in “Controlling parallelism for a query” on page 520.

## Configuration parameters for controlling parallelism

The configuration parameters that give you control over the degree of parallelism server-wide are shown in Table 22-1.

***Table 22-1: Configuration parameters for parallel execution***

| Parameter | Explanation | Comment |
|---|---|---|
| number of worker processes | The maximum number of worker processes available for all parallel queries. Each worker process requires approximately as much memory as a user connection. | Restart of server required |
| max parallel degree | The number of worker processes that can be used by a single query. It must be equal to or less than number of worker processes and equal to or greater than max scan parallel degree. | Dynamic, no restart required |
| max scan parallel degree | The maximum number of worker processes that can be used for a hash scan. It must be equal to or less than number of worker processes and max parallel degree. | Dynamic, no restart required |

Configuring number of worker processes affects the size of the data and procedure cache, so you may want to change the value of total memory also.

For more information see the *System Administration Guide*.

When you change max parallel degree or max scan parallel degree, all query plans in cache are invalidated, so the next execution of any stored procedure or trigger recompiles the plan and uses the new values.

### How limits apply to query plans

When queries are optimized, the configuration parameters affect query plans.

- max parallel degree limits:
  - The number of worker processes for a partition-based scan
  - The total combined number of worker processes for nested-loop join queries, where parallel access methods are used on more than one table
  - The number of worker processes used for the merge and sort steps in merge joins
  - The number of worker processes that can be used by parallel sort operations

- max scan parallel degree limits the number of worker processes for hash-based table scans and index scans.

## How the limits work in combination

You might configure number of worker processes to 50 to allow multiple parallel queries to operate at the same time. If the table with the largest number of partitions has 10 partitions, you might set max parallel degree to 10, limiting all select queries to a maximum of 10 worker processes. Since hash-based scans operate best with 2–3 worker processes, max scan parallel degree could be set to 3.

For a single-table query, or a join involving serial access on other tables, some of the parallel possibilities allowed by these values are:

- Parallel partition scans on any tables with 2–10 partitions
- Hash-based table scans with up to 3 worker processes
- Hash-based nonclustered index scans on tables with nonclustered indexes, with up to 3 worker processes

For nested-loop joins where parallel methods are used on more than one table, some possible parallel choices are:

- Joins using a hash-based scan on one table and partitioned-based scans on tables with 2 or 3 partitions
- Joins using partition- based scans on both tables. For example:
    - A parallel degree of 3 for a partitioned table multiplied by max scan parallel degree of 3 for a hash-based scan requires 9 worker processes.
    - A table with 2 partitions and a table with 5 partitions requires 10 worker processes for partition-based scans on both tables.
    - Tables with 4–10 partitions can be involved in a join, with one or more tables accessed in serial.

For merge joins:

- For a full-merge join, 10 worker processes scan the base tables (unless these are fewer than 10 distinct values on the join keys); the number of partitions on the tables is not considered.
- For a merge join that scans a table and selects rows into a worktable:

- The scan that precedes the merge join may be performed in serial or in parallel. The degree of parallelism is determined in the usual way for such a query.
- For the merge, 10 worker processes are used unless there are fewer distinct values in the join key.
- For the sort, up to 10 worker processes can be used.

For fast performance, while creating a clustered index on a table with 10 partitions, the setting of 50 for number of worker processes allows you to set max parallel degree to 20 for the create index command.

For more information on configuring worker processes for sorting, see "Worker process requirements for parallel sorts" on page 581.

### Examples of setting parallel configuration parameters

The following command sets number of worker processes:

```
sp_configure "number of worker processes", 50
```

After a restart of the server, these commands set the other configuration parameters:

```
sp_configure "max parallel degree", 10
sp_configure "max scan parallel degree", 3
```

To display the current settings for these parameters, use:

```
sp_configure "Parallel Query"
```

## Using *set* options to control parallelism for a session

Two set options let you restrict the degree of parallelism on a session basis or in stored procedures or triggers. These options are useful for tuning experiments with parallel queries and can also be used to restrict noncritical queries to run in serial, so that worker processes remain available for other tasks. The set options are summarized in Table 22-2.

***Table 22-2: set options for parallel execution tuning***

| Parameter | Function |
|---|---|
| parallel_degree | Sets the maximum number of worker processes for a query in a session, stored procedure, or trigger. Overrides the max parallel degree configuration parameter, but must be less than or equal to the value of max parallel degree. |

| Parameter | Function |
|---|---|
| scan_parallel_degree | Sets the maximum number of worker processes for a hash-based scan during a specific session, stored procedure, or trigger. Overrides the max scan parallel degree configuration parameter but must be less than or equal to the value of max scan parallel degree. |

If you specify a value that is too large for set either option, the value of the corresponding configuration parameter is used, and a message reports the value in effect. While set parallel_degree or set scan_parallel_degree is in effect during a session, the plans for any stored procedures that you execute are not placed in the procedure cache. Procedures executed with these options in effect may produce suboptimal plans.

### *set* command examples

This example restricts all queries started in the current session to 5 worker processes:

```
set parallel_degree 5
```

While this command is in effect, any query on a table with more than 5 partitions cannot use a partition-based scan.

To remove the session limit, use:

```
set parallel_degree 0
or
set scan_parallel_degree 0
```

To run subsequent queries in serial mode, use:

```
set parallel_degree 1
or
set scan_parallel_degree 1
```

## Controlling parallelism for a query

The parallel extension to the from clause of a select command allows users to suggest the number of worker processes used in a select statement. The degree of parallelism that you specify cannot be more than the value set with sp_configure or the session limit controlled by a set command. If you specify a higher value, the specification is ignored, and the optimizer uses the set or sp_configure limit.

The syntax for the select statement is:

```
select ...
from tablename [( [index index_name]
    [parallel [degree_of_parallelism | 1 ]]
    [prefetch size] [lru|mru] ) ]  ,
  tablename [( [index index_name]
    [parallel [degree_of_parallelism | 1]
    [prefetch size] [lru|mru] ) ]  ...
```

### Query level *parallel* clause examples

To specify the degree of parallelism for a single query, include parallel after the table name. This example executes in serial:

```
select * from huge_table (parallel 1)
```

This example specifies the index to use in the query, and sets the degree of parallelism to 2:

```
select * from huge_table (index ncix parallel 2)
```

See “Suggesting a degree of parallelism for a query” on page 419 for more information.

## Worker process availability and query execution

At runtime, if the number of worker processes specified in the query plan is not available, Adaptive Server creates an adjusted query plan to execute the query using fewer worker processes. This is called a **runtime adjustment**, and it can result in serial execution of the query.

A runtime adjustment now and then probably indicates an occasional, momentary bottleneck. Frequent runtime adjustments indicate that the system may not be configured with enough worker processes for the workload.

See “Runtime adjustments to worker processes” on page 559 for more information.

You can also use the set process_limit_action option to control whether a query or stored procedure should silently use an adjusted plan, whether it should warn the user, or whether the command should fail if it cannot use the optimal number of worker processes.

See “Using set process_limit_action” on page 569 for more information.

Runtime adjustments are transparent to end users, except:

- A query that normally runs in parallel may perform very slowly in serial.
- If set process_limit_action is in effect, they may get a warning, or the query may be aborted, depending on the setting.

## Other configuration parameters for parallel processing

Two additional configuration parameters for parallel query processing are:

- number of sort buffers – configures the maximum number of buffers that parallel sort operations can use from the data cache.

  See “Caches, sort buffers, and parallel sorts” on page 585.

- memory per worker process – establishes a pool of memory that all worker processes use for messaging during query processing. The default value, 1024 bytes per worker process, provides ample space in almost all cases, so this value should not need to be reset.

  See “Worker process management” on page 914 for information on monitoring and tuning this value.

# Commands for working with partitioned tables

Detailed steps for partitioning tables, placing them on specific devices, and loading data with parallel bulk copy are in Chapter 5, “Controlling Physical Data Placement.” The commands and tasks for creating, managing, and maintaining partitioned tables are:

- alter database – to make devices available to the database.
- sp_addsegment – to create a segment on a device; sp_extendsegment to extend the segment over additional devices, and sp_dropsegment to drop the log and system segments from data devices.
- create table...on *segment_name* – to create a table on a segment.
- alter table...partition and alter table...unpartition – to add or remove partitioning from a table.
- create clustered index – to distribute the data evenly across the table’s partitions.

- bcp (bulk copy) – with the partition number added after the table name, to copy data into specific table partitions.
- sp_helpartition – to display the number of partitions and the distribution of data in partitions, and sp_helpsegment to check the space used on each device in a segment and on the segment as a whole.

Figure 22-8 shows a scenario for creating a new partitioned table.

***Figure 22-8: Steps for creating and loading a new partitioned table***

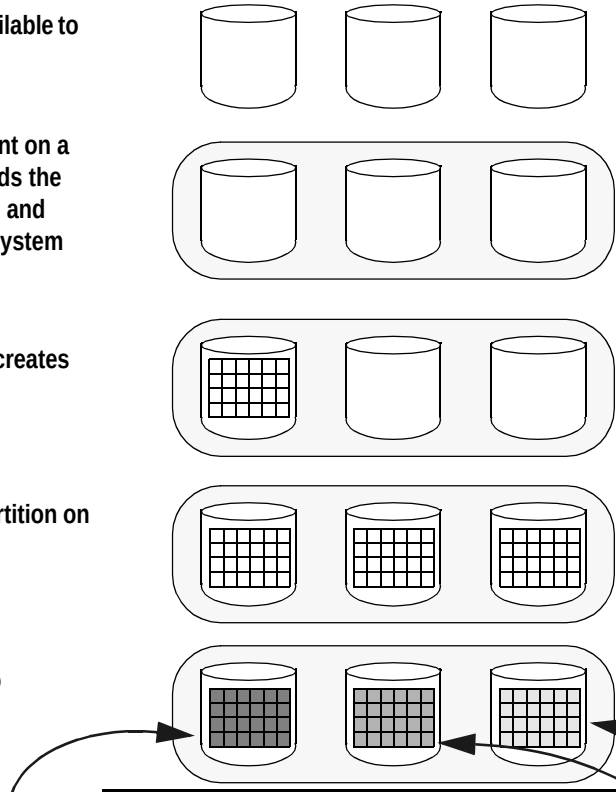
**alter database makes devices available to the database.**

**sp_addsegment creates a segment on a device, sp_extendsegment extends the segment over additional devices, and sp_dropsegment drops log and system segments from data devices.**

**create table...on *segment_name* creates the table on the segment.**

**alter table...partition creates a partition on each device.**

**Parallel bulk copy loads data into each partition from an input data file.**



| | | | |
|---|---|---|---|
| T10 | cooking | 6.95 | A Unified Approach to... |
| T10001 | cooking | 42.95 | Scheme for an internet... |
| T10007 | cooking | 47.95 | Internet Protocol Ha... |
| T10023 | cooking | 46.95 | Proposed change in P... |
| T10029 | cooking | 74.95 | System Summary for... |
| T10032 | fiction | 35.95 | Cyberpunk |
| T10035 | cooking | 49.95 | Achieving reliable coo... |
| T10038 | cooking | 12.95 | Reliable Recipes |
| T25355 | business | 69.95 | Plan and schedule |
| T39076 | psychology | 10.95 | Reallocation and Urb... |
| T56358 | UNDECIDED | 39.95 | New title |
| T75542 | romance | 44.95 | Rosalie's Romance |
| T10056 | cooking | 1.95 | Brave New Cookery |
| T25361 | business | 42.95 | Network Nuisance |
| T39082 | psychology | 6.95 | On the problem... |

authentication for network mail

# Balancing resources and performance

Maximum parallel performance requires multiple CPUs and multiple I/O devices to achieve I/O parallelism. As with most performance configuration, parallel systems reach a point of diminishing returns, and a later point where additional resources do not yield performance improvement.

You need to determine whether queries are CPU-intensive or I/O-intensive and when your performance is blocked by CPU saturation or I/O bottlenecks. If CPU utilization is low, spreading a table across more devices and using more worker processes increases CPU utilization and provides improved response time. Conversely, if CPU utilization is extremely high, but the I/O system is not saturated, increasing the number of CPUs can provide performance improvement.

## CPU resources

Without an adequate number of engines (CPU resources), tasks and worker processes must wait for access to Adaptive Server engines, and response time can be slow. Many factors determine the number of engines needed by the system, such as whether the query is CPU intensive or I/O intensive, or, at different times, both:

- Worker processes tend to spend time waiting for disk I/O and other system resources while other tasks are active on the CPU.
- Queries that perform sorts and aggregates tend to be more CPU-intensive.
- Execution classes and engine affinity bindings on parallel CPU-intensive queries can have complex effects on the system. If there are not enough CPUs, performance for both serial and parallel queries, can be degraded.

  See Chapter 4, “Distributing Engine Resources,” for more information.

## Disk resources and I/O

In most cases, configuring the physical layout of tables and indexes on devices is the key to parallel performance. Spreading partitions across different disks and controllers can improve performance during partition-based scanning if all of the following conditions are true:

- Data is distributed over different disks.
- Those disks are distributed over different controllers.
- There are enough worker processes available at runtime to allocate one worker process for each partition.

## Tuning example: CPU and I/O saturation

One experiment on a CPU-bound query found near-linear scaling in performance by adding CPUs until the I/O subsystem became saturated. At that point, additional CPU resources did not improve performance. The query performs a table scan on an 800MB table with 30 partitions, using 16K I/O. Table 22-3 shows the CPU scaling.

***Table 22-3: Scaling of engines and worker processes***

| Engines | Elapsed time, (in seconds) | CPU utilization | I/O saturation | Throughput per device, per second |
|---|---|---|---|---|
| 1 | 207 | 100% | Not saturated | .13MB |
| 2 | 100 | 98.7% | Not saturated | .27MB |
| 4 | 50 | 98% | Not saturated | .53MB |
| 8 | 27 | 93% | 100% saturated | .99MB |

# Guidelines for parallel query configuration

Parallel processing places very different demands on system resources than running the same queries in serial. Two components in planning for parallel processing are:

- A good understanding of the capabilities of the underlying hardware (especially disk drives and controllers) in use on your system
- A set of performance goals for queries you plan to run in parallel

## Hardware guidelines

Some guidelines for hardware configuration and disk I/O speeds are:

- Each Adaptive Server engine can support about five worker processes before saturating on CPU utilization for CPU-intensive queries. If CPU is not saturated at this ratio, and you want to improve parallel query performance, increase the ratio of worker processes to engines until I/O bandwidth becomes a bottleneck.
- For sequential scans, such as table scans using 16K I/O, it may be possible to achieve 1.6MB per second, per device, that is, 100 16K I/Os, or 800 pages per second, per device.
- For queries doing random access, such as nonclustered index access, the figure is approximately 50 2K I/Os, or 50 pages per second, per device.
- One I/O controller can sustain a transfer rate of up to 10–18MB per second. This means that one SCSI I/O controller can support up to 6 –10 devices performing sequential scans. Some high-end disk controllers can support more throughput. Check your hardware specifications, and use sustained rates, rather than peak rates, for your calculations.
- RAID disk arrays vary widely in performance characteristics, depending on the RAID level, the number of devices in the stripe set, and specific features, such as caching. RAID devices may provide better or worse throughput for parallelism than the same number of physical disks without striping. In most cases, start your parallel query tuning efforts by setting the number of partitions for tables on these devices to the number of disks in the array.

## Working with your performance goals and hardware guidelines

The following examples use the hardware guidelines and Table 22-3 to provide illustrate how to use parallelism to meet performance goals:

- The number of partitions for a table should be less than or equal to the number of devices. For the experiment showing scaling of engines and worker processes shown in Table 22-3, there were 30 devices available, so 30 partitions were used. Performance is optimal when each partition is placed on a separate physical device.

- Determine the number of partitions based on the I/O throughput you want to achieve. If you know your disks and controllers can sustain 1MB per second per device, and you want a table scan on an 800MB table to complete in 30 seconds, you need to achieve approximately 27MB per second total throughput, so you would need at least 27 devices with one partition per device, and at least 27 worker processes, one for each partition. These figures are very close to the I/O rates in the example in Table 22-3.
- Estimate the number of CPUs, based on the number of partitions, and then determine the optimum number by tracking both CPU utilization and I/O saturation. The example shown in Table 22-3 had 30 partitions available. Following the suggestions in the hardware guidelines of one CPU for each five devices suggests using six engines for CPU-intensive queries. At that level, I/O was not saturated, so adding more engines improved response time.

## Examples of parallel query tuning

The following examples use the I/O capabilities described in “Hardware guidelines” on page 527.

### Improving the performance of a table scan

This example shows how a table might be partitioned to meet performance goals. Queries that scan whole tables and return a limited number of rows are good candidates for parallel performance. An example is this query containing group by:

```
select type, avg(price)
    from titles
group by type
```

Here are the performance statistics and tuning goals:

| | |
|---|---|
| Table size | 48,000 pages |
| Access method | Table scan, 16K I/O |
| Serial response time | 60 seconds |
| Target performance | 6 seconds |

The steps for configuring for parallel operation are:

- Create 10 partitions for the table, and evenly distribute the data across the partitions.
- Set the number of worker processes and max parallel degree configuration parameters to at least 10.
- Check that the table uses a cache configured for 16K I/O.

In serial execution, 48,000 pages can be scanned in 60 seconds using 16K I/O. In parallel execution, each process scans 1 partition, approximately 4,800 pages, in about 6 seconds, again using 16K I/O.

### Improving the performance of a nonclustered index scan

The following example shows how performance of a query using a nonclustered index scan can be improved by configuring for a hash-based scan. The performance statistics and tuning goals are:

| | |
|---|---|
| Data pages accessed | 1500 |
| Access method | Nonclustered index, 2K I/O |
| Serial response time | 30 seconds |
| Target performance | 6 seconds |

The steps for configuring for parallel operation are:

- Set max scan parallel degree configuration parameters to 5 to use 5 worker processes in the hash-based scan.
- Set number of worker processes and max parallel degree to at least 5.

In parallel execution, each worker process scans 300 pages in 6 seconds.

## Guidelines for partitioning and parallel degree

Here are some additional guidelines to consider when you are moving from serial query execution to parallel execution or considering additional partitioning or additional worker processes for a system already running parallel queries:

- If the cache hit ratio for a table is more than 90 percent, partitioning the table will not greatly improve performance. Since most of the needed pages are in cache, there is no benefit from the physical I/O parallelism.

- If CPU utilization is more than 80 percent, and a high percentage of the queries in your system can make use of parallel queries, increasing the degree of parallelism may cause CPU saturation. This guideline also applies to moving from all-serial query processing to parallel query processing, where a large number of queries are expected to make use of parallelism. Consider adding more engines, or start with a low degree of parallelism.
- If CPU utilization is high, and a few users run large DSS queries while most users execute OLTP queries that do not operate in parallel, enabling or increasing parallelism can improve response time for the DSS queries. However, if response time for OLTP queries is critical, start with a low degree of parallelism, or make small changes to the existing degree of parallelism.
- If CPU utilization is low, move incrementally toward higher degrees of parallelism. On a system with two CPUs, and an average CPU utilization of 60 percent, doubling the number of worker processes would saturate the CPUs.
- If I/O for the devices is well below saturation, you may be able to improve performance for some queries by breaking the one-partition-per-device guideline. Except for RAID devices, always use a multiple of the number of logical devices in a segment for partitioning; that is, for a table on a segment with four devices, you can use eight partitions. Doubling the number of partitions per device may cause extra disk-head movement and reduce I/O parallelism. Creating an index on any partitioned table that has more partitions than devices prints a warning message that you can ignore in this case.

## Experimenting with data subsets

Parallel query processing can provide the greatest performance gains on your largest tables and most I/O-intensive queries. Experimenting with different physical layouts on huge tables, however, is extremely time-consuming. Here are some suggestions for working with smaller subsets of data:

- For initial exploration to determine the types of query plans that would be chosen by the optimizer, experiment with a proportional subset of your data. For example, if you have a 50-million row table that joins to a 5-million row table, you might choose to work with just one-tenth of the data, using 5 million and 500,000 rows. Select subsets of the tables that provide valid joins. Pay attention to join selectivity—if the join on the table would run in parallel because it would return 20 rows for a scan, be sure your subset reflects this join selectivity.
- The optimizer does not take underlying physical devices into account; only the partitioning on the tables. During exploratory tuning work, distributing your data on separate physical devices will give you more accurate predictions about the probable characteristics of your production system using the full tables. You can partition tables that reside on a single device and ignore any warning messages during the early stages of your planning work, such as testing configuration parameters, table partitioning and checking your query optimization. Of course, this does not provide accurate I/O statistics.

Working with subsets of data can help determine parallel query plans and the degree of parallelism for tables. One difference is that with smaller tables, sorts are performed in serial that would be performed in parallel on larger tables.

# System level impacts

In addition to other impacts described throughout this chapter, here are some concerns to be aware of when adding parallelism to mixed DSS and OLTP environments. Your goal should be improved performance of DSS through parallelism, without adverse effects on the performance of OLTP applications.

## Locking issues

Look out for lock contention:

- Parallel queries are slower than queries bench marked without contention. If the scans find many pages with exclusive locks due to update transactions, performance can change.

- If parallel queries return a large number of rows using network buffer merges, there is likely to be high contention for the network buffer. Queries hold shared locks on data pages during the scans and cause data modifications to wait for the shared locks to be released. You may need to restrict queries with large result sets to serial operation.
- If your applications experience deadlocks when DSS queries are running in serial, you may see an increase in deadlocks when you run these queries in parallel. The transaction that is rolled back in these deadlocks is likely to be the OLTP query, because the rollback decision for deadlocks is based on the accumulated CPU time of the processes involved.

  See “Deadlocks and concurrency” on page 262 for more information on deadlocks.

## Device issues

Configuring multiple devices for tempdb should improve performance for parallel queries that require worktables, including those that perform sorts and aggregates and those that use the reformatting strategy.

## Procedure cache effects

Parallel query plans are slightly larger than serial query plans because they contain extra instructions on the partition or pages that the worker processes need to access.

During ad hoc queries, each worker process needs a copy of the query plan. Space from the procedure cache is used to hold these plans in memory, and is available to the procedure cache again when the ad hoc query completes.

Stored procedures in cache are invalidated when you change the max parallel degree and max scan parallel degree configuration parameters. The next time a query is run, the query is read from disk and recompiled.

# When parallel query results can differ

When a query does not include vector or scalar aggregates or does not require a final sorting step, a parallel query might return results in a different order from the same query run in serial, and subsequent executions of the same query in parallel might return results in different order each time.

Results from serial and parallel queries that include vector or scalar aggregates, or require a final sort step, are returned after all of the results from worktables are merged or sorted in the final query processing step. Without query clauses that require this final step, parallel queries send results to the client using a network buffer merge, that is, each worker process sends results to the network buffer as it retrieves the data that satisfies the queries.

The relative speed of the different worker processes leads to differences in result set ordering. Each parallel scan behaves differently, due to pages already in cache, lock contention, and so forth. Parallel queries always return the same *set* of results, just not in the same *order*. If you need a dependable ordering of results, use order by or run the query in serial mode.

In addition, due to the pacing effects of multiple worker processes reading data pages, two types of queries accessing the same data may return different results when an aggregate or a final sort is not done:

- Queries that use set rowcount
- Queries that select a column into a local variable without sufficiently-restrictive query clauses

## Queries that use *set rowcount*

The set rowcount option stops processing after a certain number of rows are returned to the client. With serial processing, the results are consistent in repeated executions. In serial mode, the same rows are returned in the same order for a given rowcount value, because a single process reads the data pages in the same order every time.

With parallel queries, the order of the results and the set of rows returned can differ, because worker processes may access pages sooner or later than other processes. When set rowcount is in effect, each row is written to the network buffer as it is found and the buffer is sent to the client when it is full, until the required number of rows have been returned. To get consistent results, you must either use a clause that performs a final sort step or run the query in serial mode.

## Queries that set local variables

This query sets the value of a local variable in a select statement:

```
select @tid = title_id from titles
    where type = "business"
```

The where clause matches multiple rows in the titles table. so the local variable is always set to the value from the last matching row returned by the query. The value is always the same in serial processing, but for parallel query processing, the results depend on which worker process finishes last. To achieve a consistent result, use a clause that performs a final sort step, execute the query in serial mode, or add clauses so that the query arguments select only single rows.

## Achieving consistent results

To achieve consistent results for the types of queries discussed in this section, you can either add a clause to enforce a final sort or you can run the queries in serial mode. The query clauses that provide a final sort are:

- order by
- distinct, except for uses of distinct within an aggregate, such as avg(distinct price)
- union, but not union all

To run queries in serial mode, you can:

- Use set parallel_degree 1 to limit the session to serial operation
- Include the (parallel 1) clause after each table listed in the from clause of the query

CHAPTER 23

# Parallel Query Optimization

This chapter describes the basic strategies that Adaptive Server uses to perform parallel queries and explains how the optimizer applies those strategies to different queries. Parallel query optimization is an automatic process, and the optimized query plans created by Adaptive Server generally yield the best response time for a particular query.

However, knowing the internal workings of a parallel query can help you understand why queries are sometimes executed in serial, or with fewer worker processes than you expect. Knowing why these events occur can help you make changes elsewhere in your system to ensure that certain queries are executed in parallel and with the desired number of processes.

# What is parallel query optimization?

Parallel query optimization is the process of analyzing a query and choosing the best combination of parallel and serial access methods to yield the fastest response time for the query. Parallel query optimization is an extension of the serial optimization strategies discussed in earlier chapters. In addition to the costing performed for serial query optimization, parallel optimization analyzes the cost of parallel access methods for each combination of join orders, join types, and indexes. The optimizer can choose any combination of serial and parallel access methods to create the fastest query plan.

## Optimizing for response time versus total work

Serial query optimization selects the query plan that is the least costly to execute. Since only one process executes the query, choosing the least costly plan yields the fastest response time *and* requires the least amount of total work from the server.

The goal of executing queries in parallel is to get the fastest response time, even if it involves more total work from the server. During parallel query optimization, the optimizer uses cost-based comparisons similar to those used in serial optimization to select a final query plan.

However, since multiple worker processes execute the query, a parallel query plan requires more total work from Adaptive Server. Multiple worker processes, engines, and partitions that improve the speed of a query require additional costs in overhead, CPU utilization, and disk access. In other words, serial query optimization improves performance by minimizing the use of server resources, but parallel query optimization improves performance for individual queries by fully utilizing available resources to get the fastest response time.

# When is optimization performed?

The optimizer considers parallel query plans only when Adaptive Server and the current session are properly configured for parallelism, as described in “Controlling the degree of parallelism” on page 516.

If both the Adaptive Server and the current session are configured for parallel queries, then all queries within the session are eligible for parallel query optimization. Individual queries can also attempt to enforce parallel query optimization by using the optimizer hint parallel *N* for parallel or parallel 1 for serial.

If the Adaptive Server or the current session is not configured for parallel queries, or if a given query uses optimizer hints to enforce serial execution, then the optimizer considers serial access methods; the parallel access methods described in this chapter are not considered.

Adaptive Server does not execute parallel queries against system tables.

# Overhead costs

Parallel queries incur more overhead costs to perform such internal tasks as:

- Allocating and initializing worker processes
- Coordinating worker processes as they execute a query plan
- Deallocating worker processes after the query is completed

To avoid applying these overhead costs to OLTP-based queries, the optimizer "disqualifies" tables from using parallel access methods when a scan would access fewer than 20 data pages in a table. This restriction applies whether or not an index is used to access a table's data. When Adaptive Server must scan fewer than 20 data pages, the optimizer considers only serial table and index scans and does not consider parallel optimization.

## Factors that are not considered

When computing the cost of a parallel access method, the optimizer *does not* consider factors such as the number of engines available, the ratio of engines to CPUs, and whether or not a table's partitions reside on dedicated physical devices and controllers. Each of these factors can significantly affect the performance of a query. It is up to the System Administrator to ensure that these resources are configured in the best possible way for the Adaptive Server system as a whole.

See “Configuration parameters for controlling parallelism” on page 517 for information on configuring Adaptive Server.

See “Commands for partitioning tables” on page 93 for information on partitioning your data to best facilitate parallel queries.

# Parallel access methods

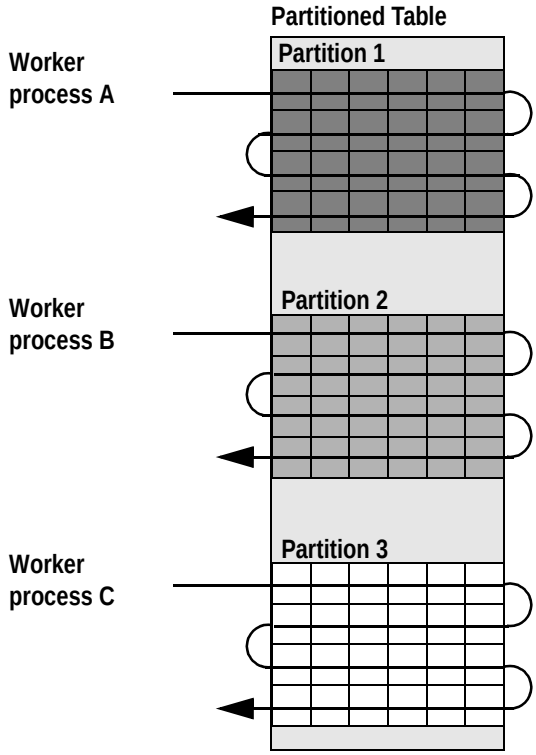
The following sections describe parallel access methods and other strategies that the optimizer considers when optimizing parallel queries. Parallel access methods fall into these general categories:

- **Partition-based access methods** use two or more worker processes to access separate partitions of a table. Partition-based methods yield the fastest response times because they can distribute the work in accessing a table over both CPUs and physical disks. At the CPU level, worker processes can be queued to separate engines to increase processing performance. At the physical disk level, worker processes can perform I/O independently of one another, if the table’s partitions are distributed over separate physical devices and controllers.
- **Hash-based access methods** provide parallel access to partitioned tables, using either table scans or index scans. Hash-based strategies employ multiple worker processes to work on a single chain of data pages or a set of index pages. I/O is not distributed over physical devices or controllers, but worker processes can still be queued to multiple engines to distribute processing and improve response times.
- **Range-based access methods** provide parallel access during merge joins on partitioned tables and unpartitioned tables, including worktables created for sorting and merging, and via indexes. The partitioning on the tables is not considered when choosing the degree of parallelism, so it is not distributed over physical devices or controllers. Worker processes can be queued to multiple engines to distribute processing and improve response times.

## Parallel partition scan

In a parallel partition scan, multiple worker processes completely scan each partition in a partitioned table. One worker process is assigned to each partition, and each process reads all pages in the partition. Figure 23-1 illustrates a parallel partition scan.

***Figure 23-1: Parallel partition scan***




The parallel partition scan operates faster than a serial table scan. The work is divided over several worker processes that can execute simultaneously on different engines. Some worker processes can be executing during the time that others sleep on I/O or other system resources. If the table partitions reside on separate physical devices, I/O parallelism is also possible.

### Requirements for consideration

The optimizer considers the parallel partition scan only for partitioned tables in a query. The table's data cannot be skewed in relation to the number of partitions, or the optimizer disqualifies partition-based access methods from consideration. Table data is considered skewed when the size of the largest partition is two or more times the average partition size.

Finally, the query must access at least 20 data pages before the optimizer considers any parallel access methods.

### Cost model

The Adaptive Server optimizer computes the cost of a parallel table partition scan as the largest number of logical and physical I/Os performed by any one worker process in the scan. In other words, the cost of this access method equals the I/O required to read all pages in the largest partition of the table.

For example, if a table with 3 partitions has 200 pages in its first partition, 300 pages in its second, and 500 pages in its last partition, the cost of performing a partition scan on that table is 500 logical and 500 physical I/Os (assuming 2K I/O for the physical I/O). In contrast, the cost of a serial scan of this table is 1000 logical and physical I/Os.

## Parallel clustered index partition scan (allpages-locked tables)

A clustered index partition scan uses multiple worker processes to scan data pages in a partitioned table when the clustered index key matches a search argument. This method can be used only on allpages-locked tables.

One worker process is assigned to each partition in the table. Each worker process accesses data pages in the partition, using one of two methods, depending on the range of key values accessed by the process. When a partitioned table has a clustered index, rows are assigned to partitions based on the clustered index key.

Figure 23-2 shows a clustered index partition scan that spans three partitions. Worker processes A, B, and C are assigned to each of the table's three partitions. The scan involves two methods:

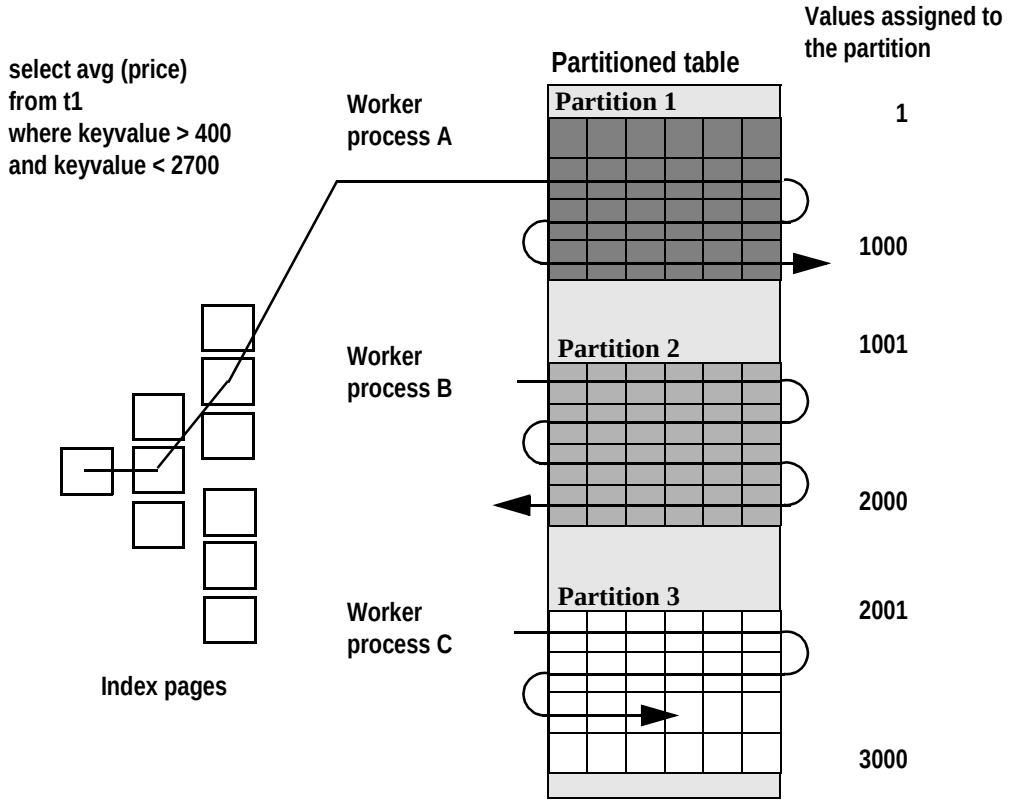
- Method 1

Worker process A traverses the clustered index to find the first starting page that satisfies the search argument, about midway through partition 1. It then begins scanning data pages until it reaches the end of partition 1.

- Method 2

  Worker processes B and C do not use the clustered index, but, instead, they begin scanning data pages from the beginning of their partitions. Worker process B completes scanning when it reaches the end of partition 2. Worker process C completes scanning about midway through partition 3, when the data rows no longer satisfy the search argument.

***Figure 23-2: Parallel clustered index partition scan***



### Requirements for consideration

The optimizer considers a clustered index partition scan only when:

- The query accesses at least 20 data pages of the table.
- The table is partitioned and uses allpages locking.
- The table's data is not skewed in relation to the number of partitions. Table data is considered skewed when the size of the largest partition is two or more times the average partition size.

### Cost model

The Adaptive Server optimizer computes the cost of a clustered index partition scan differently, depending on the total number of pages that need to be scanned:

- If the total number of pages that need to be scanned is less than or equal to two times the average size of a partition, the optimizer costs the scan as the total number of pages to be scanned divided by 2.
- If the total number of pages that need to be scanned is greater than two times the average size of a partition, the optimizer costs the scan as the average number of pages in a partition.

The actual cost of the scan may be higher if:

- The total number of pages that need to be scanned is less than the size of a partition, and
- The data to be scanned lies entirely within one partition

If both of these conditions are true, the actual cost of the scan is the same as if the scan were executed serially.

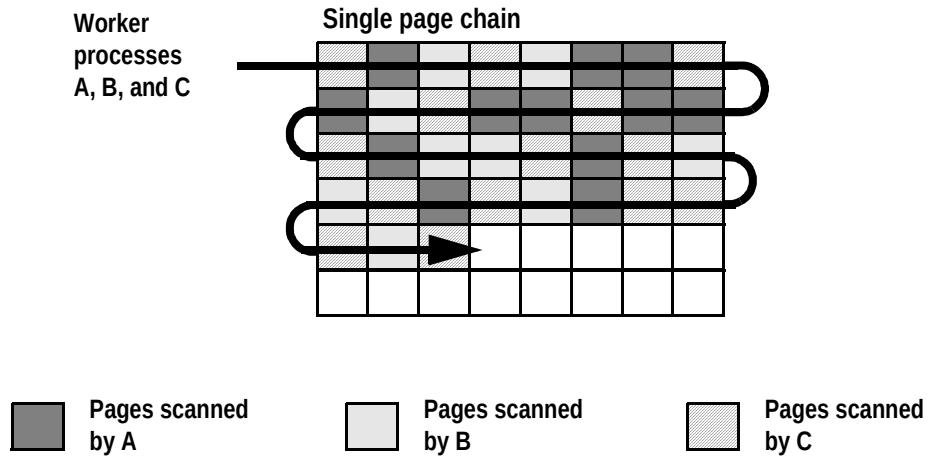
## Parallel hash-based table scan

Parallel hash-based table scans are performed slightly differently, depending on the locking scheme of the table.

## Hash-based table scans on allpages-locked tables

In a hash-based table scan on an allpages-locked table, multiple worker processes scan a single chain of data pages in a table simultaneously. All worker processes traverse the page chain and apply an internal hash function to each page ID. The hash function determines which worker process reads the rows in the current page. The hash function ensures that only one worker process scans the rows on any given page of the table. Figure 23-3 illustrates the hash-based table scan.

***Figure 23-3: Parallel hash-based table scan on an allpages-locked table***




The hash-based scan provides a way to distribute the processing of a single chain of data pages over multiple engines. The optimizer may use this access method for the outer table of a join query to process a join condition in parallel.

## Hash-based table scans on data-only-locked tables

A hash-based scan on a data-only-locked table hashes on either the extent number or the allocation page number, rather than hashing on the page number. The choice of whether to hash on the allocation page or the extent number is a cost-based decision made by the optimizer. Both methods can reduce the cost of performing parallel queries on unpartitioned tables. Queries that choose a serial scan on an allpages-locked table may use one of the new hash-based scan methods if the table is converted to data-only locking.

### Requirements for consideration

The optimizer considers the hash-based table scan only for heap tables, and only for outer tables in a join query—it does not consider this access method for clustered indexes or for single-table queries. Hash-based scans can be used on either unpartitioned or partitioned tables. The query must access at least 20 data pages of the table before the optimizer considers any parallel access methods.

### Cost model

The optimizer computes the cost of a hash-based table scan as the total number of logical and physical I/Os required to scan the table.

For an allpages-locked table, the physical I/O cost is approximately the same as for a serial table scan. The logical cost is the number of pages to be read multiplied by the number of worker processes. The cost per worker process is one logical I/O for each page in the table, and approximately 1/*N* physical I/Os, with *N* being the number of worker processes.

For a data-only-locked table, this is approximately the same cost applied to a serial table scan, with the physical and logical I/O divided evenly between the worker processes.

## Parallel hash-based index scan

An index hash-based scan can be performed using either a nonclustered index or a clustered index on a data-only-locked table. To perform the scan:

- All worker processes traverse the higher index levels.
- All worker processes scan the leaf-level index pages.

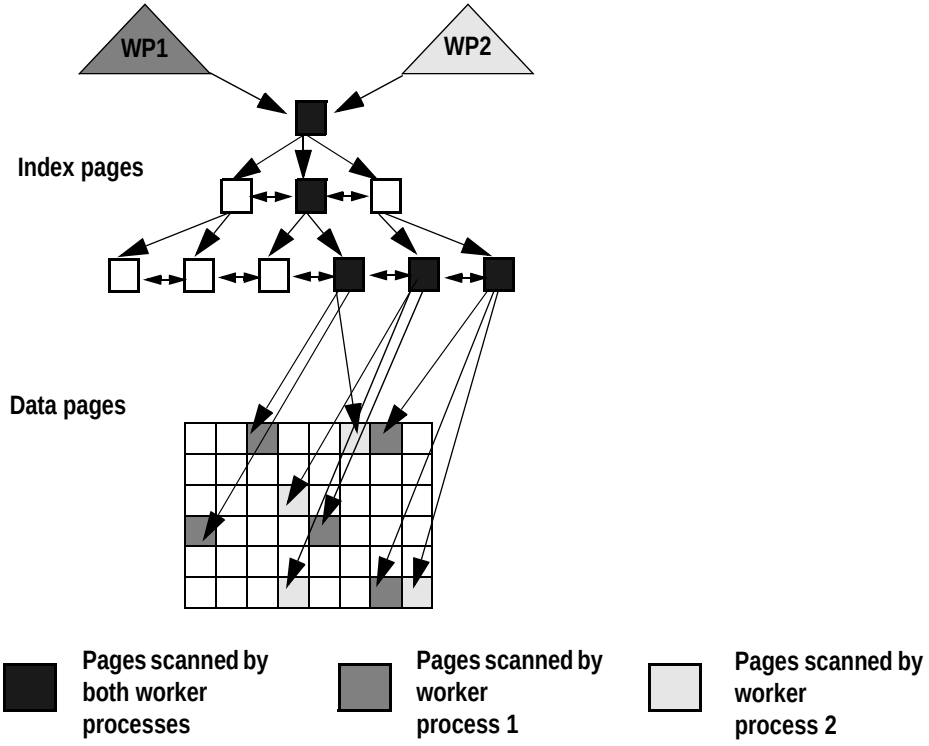
For data-only-locked tables, the worker processes scanning the leaf level hash on the page ID for each row, and scan the matching data pages.

For allpages-locked tables, a hash-based index scan is performed in one of two ways, depending on whether the table is a heap table or has a clustered index. The major difference between the two methods is the hashing mechanism:

- For a table with a clustered index, the hash is on the key values.
- For a heap table, the scan hashes on the page ID.

Figure 23-4 illustrates a nonclustered index hash-based scan on a heap table with two worker processes.

***Figure 23-4: Nonclustered index hash-based scan***




## Cost model and requirements

The cost model of a nonclustered index scan uses the formula:

Scan Cost = Number of index levels
+ Number of leaf pages / pages per IO
+ (Number of data pages / pages per IO ) / number of worker processes

The optimizer considers a hash-based index scan for any tables in a query that have useful nonclustered indexes, and for data-only-locked tables with clustered indexes. The query must also access at least 20 data pages of the table.

**Note** If a nonclustered index covers the result of a query, the optimizer does not consider using the nonclustered index hash-based scan.

See "Index covering" on page 214 for more information about index covering.

## Parallel range-based scans

Parallel range-based scans are used for the merge process in merge joins.

When two tables are merged in parallel, each worker process is assigned a range of values to merge. The range is determined using histogram statistics or sampling. When a histogram exists for at least one of the join columns, it is used to partition the ranges so that each worker process operates on approximately the same number of rows. If neither join column has a histogram, sampling similar to that performed for other parallel sort operations determines the range of values to be merged by each worker process.

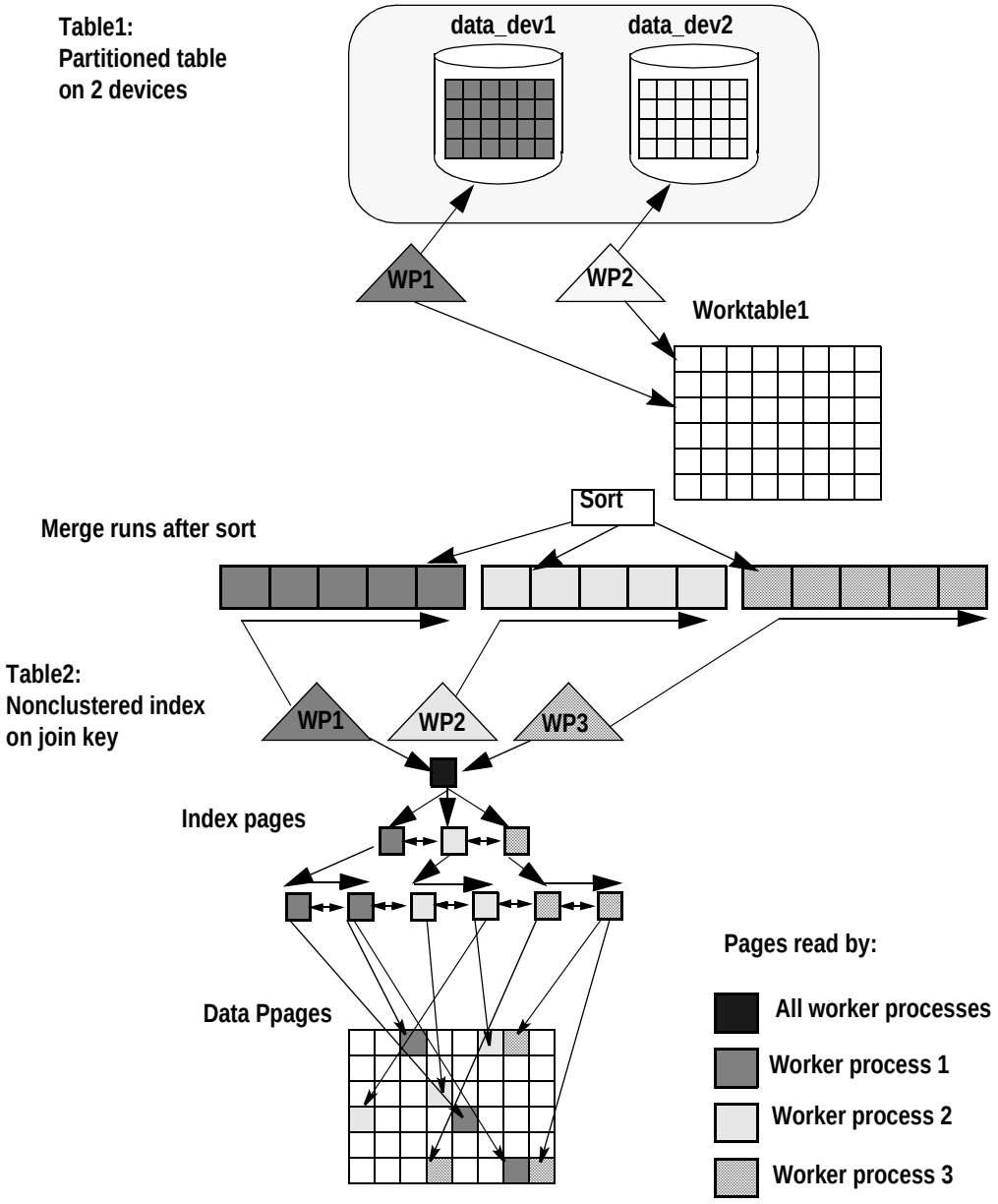
Figure 23-5 shows a parallel right-merge join. In this case:

- A right-merge join is used. Table1, the outer table, is scanned into a worktable and sorted, then merged with the inner table. These worker processes are deallocated at the end of this step.
- The outer table has two partitions, so two worker processes are used to perform a parallel partition scan.
- The inner table has a nonclustered index on the join key. max parallel degree is set to 3, so 3 worker processes are used.

### Requirements for consideration

The optimizer considers parallel merge joins when the configuration parameter enable merge joins is set to 1 and the table accesses more than 20 data pages from the outer table in the merge join.

*Figure 23-5: A parallel right-merge join*



## Additional parallel strategies

Adaptive Server may employ additional strategies when executing queries in parallel. Those strategies involve the use of partitioned worktables and parallel sorting.

### Partitioned worktables

For queries that require a worktable, Adaptive Server may choose to create a partitioned worktable and populate it using multiple worker processes. Partitioning the worktable improves performance when Adaptive Server populates the table, and therefore, improves the response time of the query as a whole.

See “Parallel query examples” on page 559 for examples of queries that can benefit from the use of partitioned worktables.

### Parallel sorting

Parallel sorting employs multiple worker processes to sort data in parallel, similar to the way multiple worker processes execute a query in parallel. create index and any query that requires sorting can benefit from the use of parallel sorting.

The optimizer does not directly optimize or control the execution of a parallel sort.

See “Parallel query examples” on page 559 for examples of queries that can benefit from the parallel sorting strategy.

Also, see “Overview of the parallel sorting strategy” on page 577 for a detailed explanation of how Adaptive Server executes a sort in parallel.

# Summary of parallel access methods

Table 23-1 summarizes the potential use of parallel access methods in Adaptive Server query processing. In all cases, the query must access at least 20 data pages in the table before the optimizer considers parallel access methods.

*Table 23-1: Parallel access method summary*

| Parallel method | Major cost factors | Requirements for consideration | Competing serial methods |
|---|---|---|---|
| Partition-based scan | Number of pages in the largest partition | Partitioned table with balanced data | Serial table scan, serial index scan |
| Hash-based table scan | Number of pages in table | Any outer table in a join query and that is a heap | Serial table scan, serial index scan |
| Clustered index partition scan | If total number of pages to be scanned <= 2 * number of pages in average-sized partition, then: Total number of pages to be scanned / 2<br>If total number of pages to be scanned > 2 * number of pages in average-sized partition, then: Average number of pages in a partition | Partitioned table with a useful clustered index; allpages locking only | Serial index scan |
| Hash-based index scan | Number of index pages above leaf level to scan + number of leaf-level index pages to scan + (number of data pages referenced in leaf-level index pages / number of worker processes) | Any table with a useful nonclustered index or a data-only-locked table with a clustered index | Serial index scan |
| Range-based scan | Number of pages to be accessed in both tables/number of worker processes, plus any sort costs | Any table in a join eligible for merge join consideration | Serial merge, nested-loop join |

## Selecting parallel access methods

For a given table in a query, the optimizer first evaluates the available indexes and partitions to determine which access methods it can use to scan the table's data. For any query that involves a join, Adaptive Server considers a range-based merge join, and considers using a parallel merge join if parallel query processing is enabled. The use of a range-based scan does not depend on table partitioning, and range-based scans can be performed using clustered and nonclustered indexes. They are considered, and are very likely to be used, on tables that have no useful index on the join key.

Table 23-2 shows the other parallel access methods that the optimizer may evaluate for different table and index combinations. Hash-based table scans are considered only for the outer table in a query, unless the query uses the parallel optimizer hint.

***Table 23-2: Determining applicable partition or hash-based access methods***

| | No useful index | Useful clustered index | Useful index (nonclustered or clustered on data-only-locked table) |
|---|---|---|---|
| Partitioned Table | Partition scan<br>Hash-based table scan (if table is a heap)<br>Serial table scan | Clustered index partition scan<br>Serial index scan | Nonclustered index hash-based scan<br>Serial index scan |
| Unpartitioned Table | Hash-based table scan (if table is a heap)<br>Serial table scan | Serial index scan | Nonclustered index hash-based scan<br>Serial index scan |

The optimizer may further eliminate parallel access methods from consideration, based on the number of worker processes that are available to the query. This process of elimination occurs when the optimizer computes the degree of parallelism for the query as a whole.

For an example, see “Partitioned heap table” on page 557.

# Degree of parallelism for parallel queries

The **degree of parallelism** for a query is the number of worker processes chosen by the optimizer to execute the query in parallel. The degree of parallelism depends on both the upper limit to the degree of parallelism for the query and on the level of parallelism suggested by the optimizer.

Computing the degree of parallelism for a query is important for two reasons:

- The final degree of parallelism directly affects the performance of a query since it specifies how many worker processes should do the work in parallel.

- While computing the degree of parallelism, the optimizer disqualifies parallel access methods that would require more worker processes than the limits set by configuration parameters, the set command, or the parallel clause in a query. This reduces the total number of access methods that the optimizer must consider when costing the query, and, therefore, decreases the overall optimization time. Disqualifying access methods in this manner is especially important for multitable joins, where the optimizer must consider many different combinations of join orders and access methods before selecting a final query plan.

## Upper limit

A System Administrator configures the upper limit to the degree of parallelism using server-wide configuration parameters. Session-wide and query-level options can further limit the degree of parallelism. These limits set both the total number of worker processes that can be used in a parallel query and the total number of worker processes that can be used for hash-based access methods.

The optimizer removes from consideration any parallel access methods that would require more worker processes than the upper limit for the query. (If the upper limit to the degree of parallelism is 1, the optimizer does not consider any parallel access methods.)

See “Configuration parameters for controlling parallelism” on page 517 for more information about configuration parameters that control the upper limit to the degree of parallelism.

## Optimized degree

The optimizer can potentially use worker processes up to the maximum degree of parallelism set at the server, session, or query level. However, the optimized degree of parallelism may be less than this maximum. For partition-based scans, the optimizer chooses the degree of parallelism based on the number of partitions in the tables of the query and the number of worker processes configured.

## Worker processes for partition-based scans

For partition-based access methods, Adaptive Server requires one worker process for every partition in a table. If the number of partitions exceeds max parallel degree or a session-level or query-level limit, the optimizer uses a hash-based or serial access method; if a merge join can be used, it may choose a merge join using the max parallel degree.

## Worker processes for hash-based scans

For hash-based access methods, the optimizer does not compute an optimal degree of parallelism; instead, it uses the number of worker processes specified by the max scan parallel degree parameter. It is up to the System Administrator to set max scan parallel degree to an optimal value for the Adaptive Server system as a whole. A general rule of thumb is to set this parameter to no more than 2 or 3, since it takes only 2–3 worker processes to fully utilize the I/O of a given physical device.

## Worker processes for range-based scans

A merge join can use multiple worker processes to perform:

- The scan that selects rows into a worktable, for any merge join that requires a sort
- The worktable sort
- The merge join and subsequent joins in the step
- The range scan of both tables during a full merge join

### Usage while creating the worktable

If a worktable is needed for a merge join, the query step that creates the worktable can use a serial or parallel access method for the scan. The number of worker processes for this step is determined by the usual methods for selecting the number of worker processes for a query. The query that selects the rows into the worktable can be a single-table query or a join performing a nested-loop or merge join, or a combination of nested-loops joins and a merge join.

#### Parallel sorting for merge-join worktables

Parallel sorting is used when the number of pages in the worktable to be sorted is eight times the value of the number of sort buffers configuration parameter.

See Chapter 24, "Parallel Sorting," for more information about parallel sorting.

#### Number of merge threads

For the merge step, the number of merge threads is set to max parallel degree, unless the number of distinct values is smaller than max parallel degree. If the number of values to be merged is smaller than the max parallel degree, the task uses one worker process per value, with each worker process merging one value. If the tables being merged have different numbers of distinct values, the lower number determines the number of worker processes to be used. The formula is:

Worker processes = min (max pll degree, min(t1_uniq_vals, t2_uniq_vals))

When there is only one distinct value on the join column, or there is an equality search argument on a join column, the merge step is performed in serial mode. If a merge join is used for this query, the merge is performed in serial mode:

```
select * from t1, t2
where t1.c1 = t2.c1
and t1.c1 = 10
```

#### Total usage for merge joins

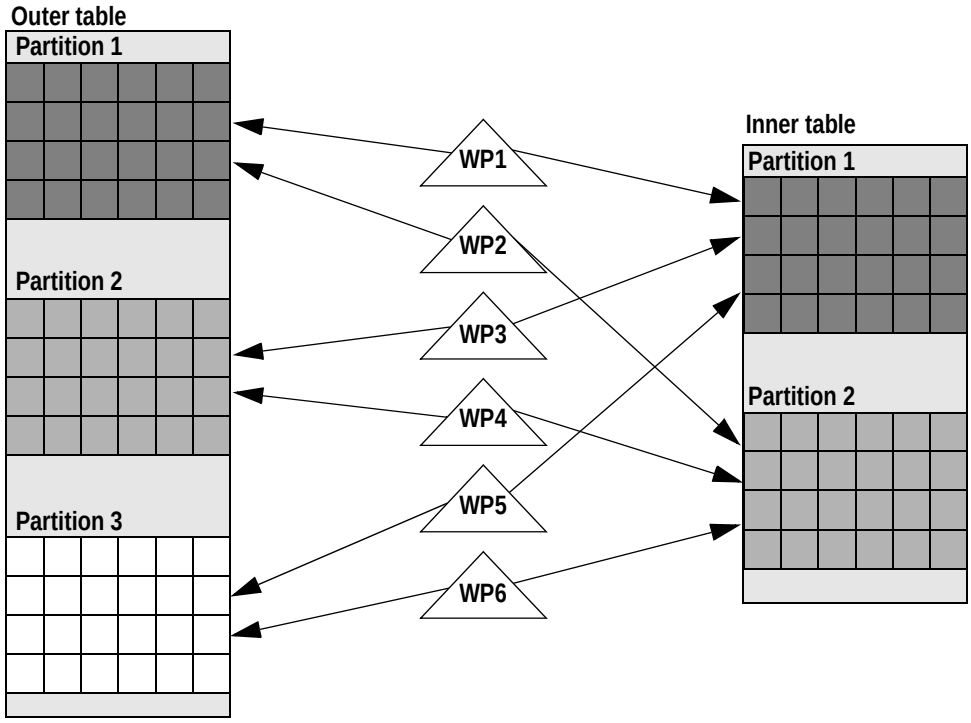
A merge join can use up to max parallel degree threads for the merge step and up to max parallel degree threads can be used for each sort. A merge that performs a parallel sort may use up to 2*max parallel degree threads. Worker processes used for sorts are released when the sort completes.

## Nested-loop joins

For individual tables in a nested-loop join, the optimizer computes the degree of parallelism using the same rules described in “Optimized degree” on page 551. However, the degree of parallelism for the join query as a whole is the *product* of the worker processes that access individual tables in the join. All worker processes allocated for a join query access all tables in the join. Using the product of worker processes to drive the degree of parallelism for a join ensures that processing is distributed evenly over partitions and that the join returns no duplicate rows.

Figure 23-6 illustrates this rule for two tables in a join where the outer table has three partitions and the inner table has two partitions. If the optimizer determines that partition-based access methods are to be used on each table, then the query requires a total of six worker processes to execute the join. Each of the six worker processes scans one partition of the outer table and one partition of the inner table to process the join condition.

***Figure 23-6: Worker process usage for a nested-loop join***




In Figure 23-6, if the optimizer chose to scan the inner table using a serial access method, only three worker processes would be required to execute the join. In this situation, each worker process would scan one partition of the outer table, and all worker processes would scan the inner table to find matching rows.

Therefore, for any two tables in a query with scan degrees of *m* and *n* respectively, the potential degrees of parallelism for a nested-loop join between the two tables are:

- 1, if the optimizer accesses both tables serially
- *m**1, if the optimizer accesses the first table using a parallel access method (with *m* worker processes), and the second table serially
- *n**1, if the optimizer accesses the second table using a parallel access method (with *n* worker processes) and the first table serially

- *m*n*, if the optimizer accesses both tables using parallel access methods

### Alternative plans

Using partition-based scans on both tables in a join is fairly rare because of the high cost of repeatedly scanning the inner table. The optimizer may also choose:

- A merge join.
- The reformatting strategy, if reformatting is a cheaper alternative.
- A partitioned-based scan plus a hash-based index scan, when a join returns rows from 20 or more data pages.

  See Figure 22-7 on page 515 for an illustration.

### Computing the degree of parallelism for nested-loop joins

To determine the degree of parallelism for a join between any two tables (and to disqualify parallel access methods that would require too many worker processes), the optimizer applies the following rules:

1 The optimizer determines possible access methods and degrees of parallelism for the outer table of the join. This process is the same as for single-table queries.

  See “Optimized degree” on page 551.

2 For each access method determined in step 1, the optimizer calculates the remaining number of worker processes that are available for the inner table of the join. The following formula determines this number:

Remaining worker processes = **max parallel degree**/ Worker processes for outer table

3 The optimizer uses the remaining number of worker processes as an upper limit to determine possible access methods and degrees of parallelism for the inner table of the join.

The optimizer repeats this process for all possible join orders and access methods and applies the cost function for joins to each combination. The optimizer selects the least costly combination of join orders and access methods, and the final combination drives the degree of parallelism for the join query as a whole.

See “Nested-loop joins” on page 554 for examples of this process.

### Parallel queries and existence joins

Adaptive Server imposes an additional restriction for subqueries processed as existence joins. For these queries, only the number of partitions in the outer table determines the degree of parallelism. There are only as many worker processes as there are partitions in the outer table. The inner table in such a query is always accessed serially. This restriction does not apply to subqueries that are flattened into regular joins.

## Examples

The examples in this section show how the limits to the degree of parallelism affect the following types of queries:

- A partition heap table
- A nonpartitioned heap table
- A table with a clustered index

### Partitioned heap table

Assume that max parallel degree is set to 10 worker processes and max scan parallel degree is set to 3 worker processes.

#### Single-table query

For a single-table query on a heap table with 6 partitions and no useful nonclustered index, the optimizer costs the following access methods:

- A parallel partition scan using 6 worker processes
- A serial table scan using a single process

If max parallel degree is set to 5 worker processes, then the optimizer does not consider the partition scan for a table with 6 partitions.

#### Query with a join

The situation changes if the query involves a join. If max parallel degree is set to 10 worker processes, the query involves a join, and a table with 6 partitions is the outer table in the query, then the optimizer considers the following access methods:

- A partition scan using 6 worker processes
- A hash-based table scan using 3 worker processes
- A merge join using 10 worker processes
- A serial scan using a single process

If max scan parallel degree is set to 5 and max scan parallel degree is set to 3, then the optimizer considers the following access methods:

- A hash-based table scan using 3 worker processes
- A merge join using 5 worker processes
- A serial scan using a single process

Finally, if max parallel degree is set to 5 and max scan parallel degree is set to 1, then the optimizer considers only a merge join as a parallel access method.

### Nonpartitioned heap table

If the query involves a join, and max scan parallel degree is set to 3, and the nonpartitioned heap table is the outer table in the query, then the optimizer considers the following access methods:

- A hash-based table scan using 3 worker processes
- A range scan using 10 worker processes for the merge join
- A serial scan using a single process

If max scan parallel degree is set to 1, then the optimizer does not consider the hash-based scan.

See “Single-table scans” on page 560 for more examples of determining the degree of parallelism for queries.

### Table with clustered index

If the table has a clustered index, the optimizer considers the following parallel access methods when the table uses allpages locking:

- A parallel partition scan or a parallel clustered index scan, if the table is partitioned and max parallel degree is set to at least 6
- A range scan, using max parallel degree worker processes
- A serial scan

If the table uses data-only-locking, the optimizer considers:

- A parallel partition scan, if the table is partitioned and max parallel degree is set to at least 6
- A range scan, using max parallel degree worker processes
- A serial scan

### Runtime adjustments to worker processes

Even after the optimizer determines a degree of parallelism for the query as a whole, Adaptive Server may make final adjustments at runtime to compensate for the actual number of worker processes that are available. If fewer worker processes are available at runtime than are suggested by the optimizer, the degree of parallelism is reduced to a level that is consistent with the available worker processes and the access methods in the final query plan. “Runtime adjustment of worker processes” on page 567 describes the process of adjusting the degree of parallelism at runtime and explains how to determine when these adjustments occur.

## Parallel query examples

The following sections further explain and provide examples of how Adaptive Server optimizes these types of parallel queries:

- Single-table scans
- Multitable joins
- Subqueries
- Queries that require worktables
- union queries
- Queries with aggregates

- select into statements

Commands that insert, delete, or update data, and commands executed from within cursors are never considered for parallel query optimization.

## Single-table scans

The simplest parallel query optimization involves queries that access a single base table. Adaptive Server optimizes these queries by evaluating the base table to determine applicable access methods, and then applying cost functions to select the least costly plan.

Understanding how Adaptive Server optimizes single-table queries is integral to understanding more complex parallel queries. Although queries such as multitable joins and subqueries use additional optimization strategies, the process of accessing individual tables for those queries is the same.

The following example shows instances in which the optimizer uses parallel access methods on single-table queries.

### Table partition scan

This example shows a query where the optimizer chooses a table partition scan over a serial table scan. The configuration and table layout are as follows:

| Configuration parameter values | |
|---|---|
| **Parameter** | **Setting** |
| max parallel degree | 10 worker processes |
| max scan parallel degree | 2 worker processes |

| Table layout | | | |
|---|---|---|---|
| **Table name** | **Useful indexes** | **Number of partitions** | **Number of pages** |
| authors | None | 5 | Partition 1: 50 pages<br>Partition 2: 70 pages<br>Partition 3: 90 pages<br>Partition 4: 80 pages<br>Partition 5: 10 pages |

The example query is:

```
select *
    from authors
    where au_lname < "L"
```

Using the logic in Table 23-2 on page 550, the optimizer determines that the following access methods are available for consideration:

- Partition scan
- Serial table scan

The optimizer does not consider a hash-based table scan for the table, since the balance of pages in the partitions is not skewed, and the upper limit to the degree of parallelism for the table, 10, is high enough to allow a partition-based scan.

The optimizer computes the cost of each access method, as follows:

Cost of table partition scan = # of pages in the largest partition = 90 pages

Cost of serial table scan = # of pages in table = 300 pages

The optimizer chooses to perform a table partition scan at a cost of 90 physical and logical I/Os. Because the table has 5 partitions, the optimizer chooses to use 5 worker processes. The final showplan output for this query is:

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Executed in parallel by coordinating process and 5 worker
processes.
    STEP 1
        The type of query is SELECT.
        Executed in parallel by coordinating process and 5
        worker processes.
    FROM TABLE
        authors
    Nested iteration.
    Table Scan.
    Forward scan.
    Positioning at start of table.
    Executed in parallel with a 5-way partition scan.
    Using I/O Size 16 Kbytes for data pages.
    With LRU Buffer Replacement Strategy for data pages.
    Parallel network buffer merge.
```

## Multitable joins

When optimizing joins, the optimizer considers the best join order for all combinations of tables and applicable access methods. The optimizer uses a different strategy to select access methods for inner and outer tables and the degree of parallelism for the join query as a whole.

As in serial processing, the optimizer weighs many alternatives for accessing a particular table. The optimizer balances the costs of parallel execution with other factors that affect join queries, such as the presence of a clustered index, the use of either nested-loop or merge joins, the possibility of reformatting the inner table, the join order, and the I/O and caching strategy. The following discussion focuses only on parallel versus serial access method choices.

### Parallel join optimization and join orders

This example illustrates how the optimizer devises a query plan for a join query that is eligible for parallel execution. The configuration and table layout are as follows:

| Configuration parameter values | |
|---|---|
| **Parameter** | **Setting** |
| max parallel degree | 15 worker processes |
| max scan parallel degree | 3 worker processes |

| Table layout | | | |
|---|---|---|---|
| **Table name** | **Number of partitions** | **Number of pages** | **Number of rows** |
| publishers | 1 (not partitioned) | 1,000 | 80,000 |
| titles | 10 | 10,000 (distributed evenly over partitions) | 800,000 |

The example query involves a simple join between these two tables:

```
select *
    from publishers, titles
    where publishers.pub_id = titles.pub_id
```

In theory, the optimizer considers the costs of all the possible combinations:

- titles as the outer table and publishers as the inner table, with titles accessed in parallel
- titles as the outer table and publishers as the inner table, with titles accessed serially
- publishers as the outer table and titles as the inner table, with titles accessed in parallel
- publishers as the outer table and titles as the inner table, with titles accessed serially
- publishers as the outer table and titles as the inner table, with publishers accessed in parallel

For example, the cost of a join order in which titles is the outer table and is accessed in parallel is calculated as follows:

The cost of having publishers as the outer table is calculated as follows:

However, other factors are often more important in determining the join order than whether a particular table is eligible for parallel access.

#### Scenario A: clustered index on publishers

The presence of a useful clustered index is often the most important factor in how the optimizer creates a query plan for a join query. If publishers has a clustered index on pub_id and titles has no useful index, the optimizer can choose the indexed table (publishers) as the inner table. With this join order, each access to the inner table takes only a few reads to find rows.

With publishers as the inner table, the optimizer costs the eligible access methods for each table. For titles, the outer table, it considers:

- A parallel partition scan (cost is number of pages in the largest partition)
- A serial table scan (cost is number of pages in the table)

For publishers, the inner table, the optimizer considers only a serial clustered index scan.

It also considers performing a merge join, sorting the worktable from titles into order on titles, either a right-merge or left-merge join.

The final cost of the query is the cost of accessing titles in parallel times the number of accesses of the clustered index on publishers.

### Scenario B: clustered index on titles

If titles has a clustered index on pub_id, and publishers has no useful index, the optimizer chooses titles as the inner table in the query.

With the join order determined, the optimizer costs the eligible access methods for each table. For publishers, the outer table, it considers:

- A hash-based table scan (the initial cost is the same as a serial table scan)

For titles, the inner table, the optimizer considers only aserial clustered index scan.

In this scenario, the optimizer chooses parallel over serial execution of publishers. Even though a hash-based table scan has the same cost as a serial scan, the processing time is cut by one-third, because each worker process can scan the inner table's clustered index simultaneously.

### Scenario C: neither table has a useful index

If neither table has a useful index, a merge join is a very likely choice for the access method. If merge joins are disabled, the table size and available cache space can be more important factors than potential parallel access for join order. The benefits of having a smaller table as the inner table outweigh the benefits of one parallel access method over the other. The optimizer chooses the publishers table as the inner table, because it is small enough to be read once and kept in cache, reducing costly physical I/O.

Then, the optimizer costs the eligible access methods for each table. For titles, the outer table, it considers:

- A parallel partition scan (cost is number of pages in the largest partition)
- A serial table scan (cost is number of pages in the table)

For publishers, the inner table, it considers only a serial table scan loaded into cache.

The optimizer chooses to access titles in parallel, because it reduces the cost of the query by a factor of 10.

In some cases where neither table has a useful index, the optimizer chooses the reformatting strategy, creating a temporary table and clustered index instead of repeatedly scanning the inner table.

## Subqueries

When a query contains a subquery, Adaptive Server uses different access methods to reduce the cost of processing the subquery. Parallel optimization depends on the type of subquery and the access methods:

- Materialized subqueries – parallel query methods are not considered for the materialization step.
- Flattened subqueries – parallel query optimization is considered only when the subquery is flattened to a regular join. It is not considered for existence joins or other flattening strategies.
- Nested subqueries – parallel access methods are considered for the outermost query block in a query containing a subquery; the inner, nested queries always execute serially. Although the optimizer considers parallel access methods for only the outermost query block in a subquery, all worker processes that access the outer query block also access the inner tables of the nested subqueries.

  Each worker process accesses the inner, nested query block in serial. Although the subquery is run once for each row in the outer table, each worker process performs only one-fifth of the executions. showplan output for the subquery indicates that the nested query is "Executed by 5 worker processes," since each worker process used in the outer query block scans the table specified in the inner query block.

  Each worker process maintains a separate cache of subquery results, so the subquery may be executed slightly more often than in serial processing.

## Queries that require worktables

Parallel queries that require worktables create partitioned worktables and populate them in parallel. For queries that require sorts, the parallel sort manager determines whether to use a serial or parallel sort.

See Chapter 24, "Parallel Sorting," for more information about parallel sorting.

## *union* queries

The optimizer considers parallel access methods for each part of a union query separately. Each select in a union is optimized separately, so one query can use a parallel plan, another a serial plan, and a third a parallel plan with a different number of worker processes. If a union query requires a worktable, then the worktable may also be partitioned and populated in parallel by worker processes.

If a union query is to return no duplicate rows, then a parallel sort may be performed on the internal worktable to remove duplicate rows.

See Chapter 24, "Parallel Sorting," for more information about parallel sorting.

## Queries with aggregates

Adaptive Server considers parallel access methods for queries that return aggregate results in the same way it does for other queries. For queries that use the group by clause to return a grouped aggregate result, Adaptive Server also creates multiple worktables with clustered indexes—one worktable for each worker process that executes the query. Each worker process stores partial aggregate results in its designated worktable. As worker processes finish computing their partial results, they merge those results into a common worktable. After all worker processes have merged their partial results, the common worktable contains the final grouped aggregate result set for the query.

## *select into* statements

select into creates a new table to store the query's result set. Adaptive Server optimizes the base query portion of a select into command in the same way it does a standard query, considering both parallel and serial access methods. A select into statement that is executed in parallel:

1 Creates the new table using columns specified in the select into statement.

2 Creates *n* partitions in the new table, where *n* is the degree of parallelism that the optimizer chose for the query as a whole.

3 Populates the new table with query results, using *n* worker processes.

4 Unpartitions the new table.

Performing a select into statement in parallel requires additional steps than the equivalent serial query plan. Therefore, the execution of a parallel select into statement takes place using four discrete transactions, rather than the two transactions of a serial select into statement. See select in the *Adaptive Server Reference Manual* for information about how this affects the database recovery process.

# Runtime adjustment of worker processes

The output of showplan describes the optimized plan for a given query. An optimized query plan specifies the access methods and the degree of parallelism that the optimizer suggests when the query is compiled. At execution time, there may be fewer worker processes available than are required by the optimized query plan. This can occur when:

- There are not enough worker processes available for the optimized query plan.
- The server-level or session-level limits for the query were reduced after the query was compiled. This can happen with queries executed from within stored procedures.

In these circumstances, Adaptive Server may create an adjusted query plan to compensate for the available worker processes. An **adjusted query plan** is generated at runtime and compensates for the lack of available worker processes. An adjusted query plan may use fewer worker processes than the optimized query plan, and it may use a serial access method instead of a parallel method for one or more of the tables.

The response time of an adjusted query plan may be significantly longer than its optimized counterpart. Adaptive Server provides:

- A set option, process_limit_action, which allows you to control whether runtime adjustments are allowed.
- Information on runtime adjustments in sp_sysmon output.

## How Adaptive Server adjusts a query plan

Adaptive Server uses two basic rules to reduce the number of required worker processes in an adjusted query plan:

1 If the optimized query plan specifies a partition-based access method for a table, but not enough processes are available to scan each partition, the adjusted plan uses a serial access method.

2 If the optimized query plan specifies a hash-based access method for a table, but not enough processes are available to cover the optimized degree of parallelism, the adjusted plan reduces the degree of parallelism to a level consistent with the available worker processes.

To illustrate the first case, assume that an optimized query plan recommends scanning a table's five partitions using a partition-based table scan. If only four worker processes are actually available at the time the query executes, Adaptive Server creates an adjusted query plan that accesses the table in serial, using a single process.

In the second case, if the optimized query plan recommended scanning the table with a hash-based access method and five worker processes, the adjusted query plan would still use a hash-based access method, but with, at the most, four worker processes.

## Evaluating the effect of runtime adjustments

Although optimized query plans generally outperform adjusted query plans, the difference in performance is not always significant. The ultimate effect on performance depends on the number of worker processes that Adaptive Server uses in the adjusted plan, and whether or not a serial access method is used in place of a parallel method. Obviously, the most negative impact on performance occurs when Adaptive Server uses a serial access method instead of a parallel access method to execute a query.

The performance of multitable join queries can also suffer dramatically from adjusted query plans, since Adaptive Server does not change the join ordering when creating an adjusted query plan. If an adjusted query plan is executed in serial, the query can potentially perform more slowly than an optimized serial join. This may occur because the optimized parallel join order for a query is different from the optimized serial join order.

## Recognizing and managing runtime adjustments

Adaptive Server provides two mechanisms to help you observe runtime adjustments of query plans.

- set process_limit_action allows you to abort batches or procedures when runtime adjustments take place or print warnings.
- showplan prints an adjusted query plan when runtime adjustments occur, and showplan is effect.

### Using *set process_limit_action*

The process_limit_action option to the set command lets you monitor the use of adjusted query plans at a session or stored procedure level. When you set process_limit_action to "abort," Adaptive Server records Error 11015 and aborts the query, if an adjusted query plan is required. When you set process_limit_action to "warning," Adaptive Server records Error 11014 but still executes the query.

For example, this command aborts the batch when a query is adjusted at runtime:

```
set process_limit_action abort
```

By examining the occurrences of Errors 11014 and 11015 in the error log, you can determine the degree to which Adaptive Server uses adjusted query plans instead of optimized query plans. To remove the restriction and allow runtime adjustments, use:

```
set process_limit_action quiet
```

See set in the *Adaptive Server Reference Manual* for more information about process_limit_action.

### Using *showplan*

When you use showplan, Adaptive Server displays the optimized plan for a given query before it runs the query. When the query plan involves parallel processing, and a runtime adjustment is made, showplan displays this message, followed by the adjusted query plan:

```
AN ADJUSTED QUERY PLAN WILL BE USED FOR STATEMENT 1
BECAUSE NOT ENOUGH WORKER PROCESSES ARE AVAILABLE AT
THIS TIME.
```

Adaptive Server does not attempt to execute a query when the set noexec is in effect, so runtime plans are never displayed while using this option.

## Reducing the likelihood of runtime adjustments

To reduce the number of runtime adjustments, you must increase the number of worker processes that are available to parallel queries. You can do this either by adding more total worker processes to the system or by restricting or eliminating parallel execution for noncritical queries, as follows:

- Use set parallel_degree and/or set scan_parallel_degree to set session-level limits on the degree of parallelism, or
- Use the query-level parallel 1 and parallel *N* clauses to limit the worker process usage of individual statements.

To reduce the number of runtime adjustments for system procedures, recompile the procedures after changing the degree of parallelism at the server or session level. See sp_recompile in the *Adaptive Server Reference Manual* for more information.

## Checking runtime adjustments with *sp_sysmon*

sp_sysmon shows how many times a request for worker processes was denied due to a lack of worker processes and how many times the number of worker processes recommended for a query was adjusted to a smaller number. The following sections of the report provide information:

- "Worker process management" on page 914 describes the output for the number of worker process requests that were requested and denied and the success and failure of memory requests for worker processes.
- "Parallel query management" on page 917 describes the sp_sysmon output that reports on the number of runtime adjustments and locks for parallel queries.

If insufficient worker processes in the pool seems to be the problem, compare the number of worker processes used to the number of worker processes configured. If the maximum number of worker processes used is equal to the configured value for number of worker processes, and the percentage of worker process requests denied is greater than 80 percent, increase the value for number of worker processes and re-run sp_sysmon. If the maximum number of worker processes used is less than the configured value for number of worker processes, and the percentage of worker thread requests denied is 0 percent, decreases the value for number of worker processes to free memory resources.

# Diagnosing parallel performance problems

The following sections provide troubleshooting guidelines for parallel queries. They cover two situations:

- The query runs in serial, when you expect it to run in parallel.
- The query runs in parallel, but does not perform as well as you expect.

## Query does not run in parallel

If you think that a query should run in parallel but does not, possible explanations are:

- The max parallel degree configuration parameter is set to 1, or the session-level setting set parallel_degree is set to 1, preventing all parallel access.
- The max scan parallel degree configuration parameter is set to 1, or the session level setting set scan_parallel_degree is set to 1, preventing hash-based parallel access.
- There are insufficient worker threads at execution time. Check for runtime adjustments, using the tools discussed in “Runtime adjustments to worker processes” on page 559.
- The scope of the scan is less than 20 data pages. This can be bypassed with the (parallel) clause.
- The plan calls for a table scan and:

  - The table is not a heap,
  - The table is not partitioned,
  - The partitioning is unbalanced, or
  - The table is a heap but is not the outer table of a join.

  The last two conditions can be bypassed with the (parallel) clause.
- The plan calls for a clustered index scan and:
  - The table is not partitioned, or
  - The partitioning is unbalanced. This can be bypassed with the (parallel) clause.
- The plan calls for a nonclustered index scan, and the chosen index covers the required columns.
- The table is a temporary table or a system table.
- The table is the inner table of an outer join.
- A limit has been set through the Resource Governor, and all parallel plans exceed that limit in terms of total work.
- The query is a type that is not made parallel, such as an insert, update, or delete command, a nested (not the outermost) query, or a cursor.

## Parallel performance is not as good as expected

Possible explanations are:

- There are too many partitions for the underlying physical devices.
- There are too many devices per controller.
- The (parallel) clause has been used inappropriately.
- The max scan parallel degree is set too high; the recommended range is 2–3.

## Calling technical support for diagnosis

If you cannot diagnose the problem using these hints, the following information will be needed by Sybase Technical Support to determine the source of the problem:

- The table and index schema—create table, alter table...partition, and create index statements are most helpful. Provide output from sp_help if the actual create and alter commands are not available.
- The query.
- The output of the query run with commands:
  - dbcc traceon (3604,302, 310)
  - set showplan on
  - set noexec on
- The statistics io output for the query.

# Resource limits for parallel queries

The tracking of I/O cost limits may be less precise for partitioned tables than for unpartitioned tables, when Adaptive Server is configured for parallel query processing.

When you query a partitioned table, all the labor in processing the query is divided among the partitions. For example, if you query a table with three partitions, the query's work is divided among 3 worker processes. If the user has specified an I/O resource limit with an upper bound of 6000, the optimizer assigns a limit of 2000 to each worker process.

However, since no two threads are guaranteed to perform the exact same amount of work, the parallel processor cannot precisely distribute the work among worker processes. You may get an error message saying you have exceeded your I/O resource limit when, according to showplan or statistics io output, you actually have not. Conversely, one partition may exceed the limit slightly, without the limit taking effect.

See the *System Administration Guide* for more information about setting resource limits.

CHAPTER 24

# Parallel Sorting

This chapter discusses how to configure the server for improved performance for commands that perform parallel sorts.

The process of sorting data is an integral part of any database management system. Sorting is for creating indexes and for processing complex queries. The Adaptive Server parallel sort manager provides a high-performance, parallel method for sorting data rows. All Transact-SQL commands that require an internal sort can benefit from the use of parallel sorting.

Parallel sorting and how it works and what factors affect the performance of parallel sorts is also covered. You need to understand these subjects to get the best performance from parallel sorting, and to keep parallel sort resource requirements from interfering with other resource needs.



## Commands that benefits from parallel sorting

Any Transact-SQL command that requires data row sorting can benefit from parallel sorting techniques. These commands are:

- create index commands and the alter table...add constraint commands that build indexes, unique and primary key
- Queries that use the order by clause
- Queries that use distinct

- Queries that perform merge joins requiring sorts
- Queries that use union (except union all)
- Queries that use the **reformatting strategy**

In addition, any cursors that use the above commands can benefit from parallel sorting.

# Requirements and resources overview

Like parallel query processing, parallel sorting requires more resources than performing the same command in parallel. Response time for creating the index or sorting query results improves, but the server performs more work due to overhead.

Adaptive Server's sort manager determines whether the resources required to perform a sort operation in parallel are available, and also whether a serial or parallel sort should be performed, given the size of the table and other factors. For a parallel sort to be performed, certain criteria must be met:

- The select into/bulk copy/pllsort database option must be set to true with sp_dboption in the target database:
  - For indexes, the option must be enabled in the database where the table resides. For creating a clustered index on a partitioned table, this option must be enabled, or the sort fails. For creating other indexes, serial sorts can be performed if parallel sorts cannot be performed.
  - For sorting worktables, this option must be on in tempdb. Serial sorts can be performed if parallel sorts cannot be performed.
- Parallel sorts must have a minimum number of worker processes available. The number depends on the number of partitions on the table and/or the number of devices on the target segment. The degree of parallelism at the server and session level must be high enough for the sort to use at least the minimum number of worker processes required for a parallel sort. Clustered indexes on partitioned tables must be created in parallel; other sorts can be performed in serial if there are not enough worker processes available. "Worker process requirements for parallel sorts" on page 581 and "Worker process requirements for select query sorts" on page 584.

- For select commands that require sorting, and for creating nonclustered indexes, the table to be sorted must be at least eight times the size of the available sort buffers (the value of the number of sort buffers configuration parameter), or the sort will be performed in serial mode. This ensures that Adaptive Server does not perform parallel sorting on smaller tables that would not show significant improvements in performance. This rule does not apply to creating clustered indexes on partitioned tables, since this operation always requires a parallel sort.

  See "Sort buffer configuration guidelines" on page 587.

- For create index commands, the value of the number of sort buffers configuration parameter must be at least as large as the number of worker processes available for the parallel sort.

  See "Sort buffer configuration guidelines" on page 587.

**Note** You cannot use the dump transaction command after indexes are created using a parallel sort. You must dump the database. Serial create index commands can be recovered, but only by completely re-doing the indexing command, which can greatly lengthen recovery time. Performing database dumps after serial create indexes is recommended to speed recovery, although it is not required in order to use dump transaction.

# Overview of the parallel sorting strategy

Like the Adaptive Server optimizer, the Adaptive Server parallel sort manager analyzes the available worker processes, the input table, and other resources to determine the number of worker processes to use for the sort.

After determining the number of worker processes to use, Adaptive Server executes the parallel sort. The process of executing a parallel sort is the same for create index commands and queries that require sorts. Adaptive Server executes a parallel sort by:

1 Creating a distribution map. For a merge join with statistics on a join column, histogram statistics are used for the distribution map. In other cases, the input table is sampled to build the map.

2 Reading the table data and dynamically partitioning the key values into a set of sort buffers, as determined by the distribution map.

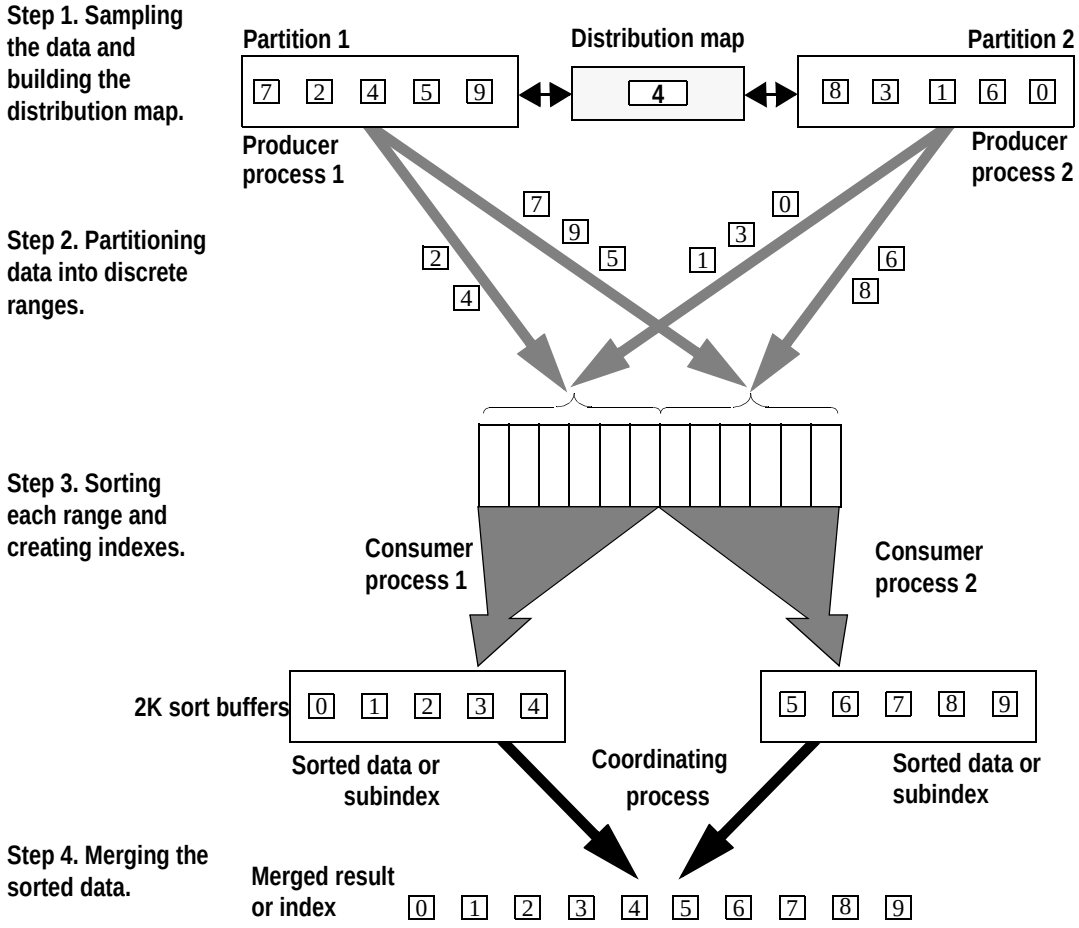
3 Sorting each individual range of key values and creating subindexes.

4 Merging the sorted subindexes into the final result set.

Each of these steps is described in the sections that follow.

Figure 24-1 depicts a parallel sort of a table with two partitions and two physical devices on its segment.

***Figure 24-1: Parallel sort strategy***



## Creating a distribution map

As a first step in executing a parallel sort, Adaptive Server creates a distribution map. If the sort is performed as part of a merge join, and there are statistics on the join columns, the histograms are used to build the distribution map. For other sorts, Adaptive Server selects and sorts a random sample of data from the input table. This distribution information—referred to as the distribution map—is used in the second sort step to divide the input data into equally sized ranges during the next phase of the parallel sort process.

The distribution map contains a key value for the highest key that is assigned to each range, except the final range in the table. In Figure 24-1, the distribution map shows that all values less than or equal to 4 are assigned to the first range and that all values greater than 4 are assigned to the second range.

## Dynamic range partitioning

After creating the distribution map, Adaptive Server employs two kinds of worker processes to perform different parts of the sort. These worker processes are called **producer processes** and **consumer processes**:

- Producer processes read data from the input table and use the distribution map to determine the range to which each key value belongs. The producers distribute the data by copying it to the sort buffers belonging to the correct range.
- Each consumer process reads the data from a range of the sort buffers and sorts it into subindexes, as described in “Range sorting” on page 580.

In Figure 24-1, two producer processes read data from the input table. Each producer process scans one table partition and distributes the data into ranges using the distribution map. For example, the first producer process reads data values 7, 2, 4, 5, and 9. Based on the information in the distribution map, the process distributes values 2 and 4 to the first consumer process, and values 7, 5, and 9 to the second consumer process.

## Range sorting

Each partitioned range has a dedicated consumer process that sorts the data in that range independently of other ranges. Depending on the size of the table and the number of buffers available to perform the sort, the consumers may perform multiple merge runs, writing intermediate results to disk, and reading and merging those results, until all of the data for the assigned range is completely sorted.

- For create index commands, each consumer for each partitioned range of data writes to a separate database device. This improves performance through increased I/O parallelism, if database devices reside on separate physical devices and controllers. The consumer process also builds an index, referred to as a subindex, on the sorted data.
- For merge joins, each consumer process writes the ordered rows to a separate set of linked data pages, one for each worker process that will perform the merge.
- For queries, the consumer process simply orders the data in the range from the smallest value to the largest.

## Merging results

After all consumer processes have finished sorting the data for each partitioned range:

- For create index commands, the coordinating process merges the subindexes into one final index.
- For merge joins, the worker processes for the merge step perform the merge with the other tables in the merge join.
- For other queries, the coordinating process merges the sort results and returns them to the client.

# Configuring resources for parallel sorting

The following sections describe the resources used by Adaptive Server when sorting data in parallel:

- Worker processes read the data and perform the sort.
- Sort buffers pass data in cache from producers to consumers, reducing physical I/O.
- Large I/O pools in the cache used for the sort also help reduce physical I/O.
- Multiple physical devices increase I/O parallelism and help determine the number of worker processes for most sorts.

## Worker process requirements for parallel sorts

Adaptive Server requires a minimum number of worker processes to perform a parallel sort. If additional worker processes are available, the sort can be performed more quickly. The minimum number required and the maximum number that can be used are determined by the number of:

- Partitions on the table, for creating clustered indexes
- Devices, for creating nonclustered indexes
- Threads used to create the worktable and the number of devices in tempdb, for merge joins
- Devices in tempdb, for other queries that require sorts

If the minimum number of worker processes is not available:

- Sorts for clustered indexes on partitioned tables must be performed in parallel; the sort fails if not enough worker processes are available.
- Sorts for nonclustered indexes and sorts for clustered indexes on unpartitioned tables can be performed in serial.
- All sorts for queries can be performed in serial.

The availability of worker processes is determined by server-wide and session-wide limits. At the server level, the configuration parameters number of worker processes and max parallel degree limit the total size of the pool of worker processes and the maximum number that can be used by any create index or select command.

The available processes at runtime may be smaller than the configured value of max parallel degree or the session limit, due to other queries running in parallel. The decision on the number of worker processes to use for a sort is made by the sort manager, not by the optimizer. Since the sort manager makes this decision at runtime, parallel sort decisions are based on the actual number of worker processes available when the sort begins.

See “Controlling the degree of parallelism” on page 516 for more information about controlling the server-wide and session-wide limits.

## Worker process requirements for creating indexes

Table 24-1 shows the number of producers and consumers required to create indexes. The **target segment** for a sort is the segment where the index is stored when the create index command completes. When you create an index, you can specify the location with the on *segment_name* clause. If you do not specify a segment, the index is stored on the default segment.

***Table 24-1: Number of producers and consumers used for create index***

| Index type | Producers | Consumers |
|---|---|---|
| Nonclustered index | Number of partitions, or 1 | Number of devices on target segment |
| Clustered index on unpartitioned table | 1 | Number of devices on target segment |
| Clustered index on partitioned table | Number of partitions, or 1 | Number of partitions |

Consumers are the workhorses of parallel sort, using CPU time to perform the actual sort and using I/O to read and write intermediate results and to write the final index to disk. First, the sort manager assigns one worker process as a consumer for each target device. Next, if there are enough available worker processes, the sort manager assigns one producer to each partition in the table. If there are not enough worker processes to assign one producer to each partition, the entire table is scanned by a single producer.

### Clustered indexes on partitioned tables

To create a clustered index on a partitioned table, Adaptive Server requires at least one consumer process for every partition on the table, plus one additional worker process to scan the table. If fewer worker processes are available, then the create clustered index command fails and prints a message showing the available and required numbers of worker processes.

If enough worker processes are available, the sort manager assigns one producer process per partition, as well as one consumer process for each partition. This speeds up the reading of the data.

| | |
|---|---|
| Minimum | 1 consumer per partition, plus 1 producer |
| Maximum | 2 worker processes per partition |
| Can be performed in serial | No |

### Clustered indexes on unpartitioned tables

Only one producer process can be used to scan the input data for unpartitioned tables. The number of consumer processes is determined by the number of devices on the segment where the index is to be stored. If there are not enough worker processes available, the sort can be performed in serial.

| | |
|---|---|
| Minimum | 1 consumer per device, plus 1 producer |
| Maximum | 1 consumer per device, plus 1 producer |
| Can be performed in serial | Yes |

### Nonclustered indexes

The number of consumer processes is determined by the number of devices on the target segment. If there are enough worker processes available and the table is partitioned, one producer process is used for each partition on the table; otherwise, a single producer process scans the entire table. If there are not enough worker processes available, the sort can be performed in serial.

| | |
|---|---|
| Minimum | 1 consumer per device, plus 1 producer |
| Maximum | 1 consumer per device, plus 1 producer per partition |
| Can be performed in serial | Yes |

### Using *with consumers* while creating indexes

RAID devices appear to Adaptive Server as a single database device, so, although the devices may be capable of supporting the I/O load of parallel sorts, Adaptive Server assigns only a single consumer for the device, by default.

The with consumers clause to the create index statement provides a way to specify the number of consumer processes that create index can use. By testing the I/O capacity of striped devices, you can determine the number of simultaneous processes your RAID device can support and use this number to suggest a degree of parallelism for parallel sorting. As a baseline, use one consumer for each underlying physical device. This example specifies eight consumers:

```
create index order_ix on orders (order_id)
with consumers = 8
```

You can also use the with consumers clause with the alter table...add constraint clauses that create the primary key and unique indexes:

```
alter table orders
add constraint prim_key primary key (order_id) with
consumers = 8
```

The with consumers clause can be used for creating indexes—you cannot control the number of consumer processes used in internal sorts for parallel queries. You cannot use this clause when creating a clustered index on a partitioned table. When creating a clustered index on a partitioned table, Adaptive Server must use one consumer process for every partition in the table to ensure that the final, sorted data is distributed evenly over partitions.

Adaptive Server ignores the with consumers clause if the specified number of processes is higher than the number of available worker processes, or if the specified number of processes exceeds the server or session limits for parallelism.

## Worker process requirements for *select* query sorts

Queries that require worktable sorts have multistep query plans. The determination of the number of worker processes for a worktable sort is made after the scan of the base table completes. During the phase of the query where data is selected into the worktable, each worker process selects data into a separate partition of the worktable.

Once the worktable is populated, additional worker processes are allocated to perform the sort step. showplan does not report this value; the sort manager reports only whether the sort is performed in serial or parallel. The worker processes used in the previous step do not participate in the sort, but remain allocated to the parallel task until the task completes.

### Worker processes for merge-join sorts

For merge joins, one consumer process is assigned for each device in tempdb; if there is only one device in tempdb, two consumer processes are used. The number of producers depends on the number of partitions in the worktable, and the setting for max parallel degree:

- If the worktable is not partitioned, one producer process is used.
- If the number of consumers plus the number of partitions in the worktable is less than or equal to max parallel degree, one producer process is allocated for each worktable partition.
- If the number of consumer processes plus the number of partitions in the worktable is greater than max parallel degree, one producer process is used.

### Other worktable sorts

For all other worktable sorts, the worktable is unpartitioned when the step that created it completes. Worker processes are assigned in the following way:

- If there is only one device in tempdb, the sort is performed using two consumers and one producer; otherwise, one consumer process is assigned for each device in tempdb, and a single producer process scans the worktable.
- If there are more devices in tempdb than the available worker processes when the sort starts, the sort is performed in serial.

## Caches, sort buffers, and parallel sorts

Optimal cache configuration and an optimal setting for the number of sort buffers configuration parameter can greatly speed the performance of parallel sorts. The tuning options to consider when you work with parallel sorting are:

- Cache bindings
- Sort buffers
- Large I/O

In most cases, the configuration you choose for normal runtime operation should be aimed at the needs of queries that perform worktable sorts. You need to understand how many simultaneous sorts are needed and the approximate size of the worktables, and then configure the cache used by tempdb to optimize the sort.

If you drop and create indexes during periods of low system usage, you can reconfigure caches and pools and change cache bindings to optimize the sorts and reduce the time required. If you need to perform index maintenance while users are active, you need to consider the impact that re configuration could have on user response time. Configuring a large percentage of the cache for exclusive use by the sort or temporarily unbinding objects from caches can seriously impact performance for other tasks.

## Cache bindings

Sorts for create index take place in the cache to which the table is bound. If the table is not bound to a cache, but the database is, then cache is used. If there is no explicit cache binding, the default data cache is used. Worktable sorts use the cache to which tempdb is bound, or the default data cache.

To configure the number of sort buffers and large I/O for a particular sort, always check the cache bindings. You can see the binding for a table with sp_help. To see all of the cache bindings on a server, use sp_helpcache. Once you have determined the cache binding for a table, use sp_cacheconfig check the space in the 2K and 16K pools in the cache.

## Number of sort buffers can affect sort performance

Producers perform disk I/O to read the input table, and consumers perform disk I/O to read and write intermediate sort results to and from disk. During the sort, producers pass data to consumers using the sort buffers. This avoids disk I/O by copying data rows completely in memory. The reserved buffers are not available to any other tasks for the duration of the sort.

The number of sort buffers configuration parameter determines the maximum space that can be used to perform a serial sort. Each sort instance can use up to the number of sort buffers value for each sort. If active sorts have reserved all of the buffers in a cache, and another sort needs sort buffers, that sort waits until buffers are available in the cache.

## Sort buffer configuration guidelines

Since number of sort buffers controls the amount of data that can be read and sorted in one batch, configuring more sort buffers increases the batch size, reduces the number of merge runs needed, and makes the sort run faster. Changing number of sort buffers is dynamic, so you do not have to restart the server.

Some general guidelines for configuring sort buffers are as follows:

- The sort manager chooses serial sorts when the number of pages in a table is less than 8 times the value of number of sort buffers. In most cases, the default value (500) works well for select queries and small indexes. At this setting, the sort manager chooses serial sorting for all create index and worktable sorts of 4000 pages or less, and parallel sorts for larger result sets, saving worker processes for query processing and larger sorts. It allows multiple sort processes to use up to 500 sort buffers simultaneously.

  A temporary worktable would need to be very large before you would need to set the value higher to reduce the number of merge runs for a sort. See “Sizing the tempdb” on page 623 for more information.

- If you are creating indexes on large tables while other users are active, configure the number of sort buffers so that you do not disrupt other activity that needs to use the data cache.

- If you are re-creating indexes during scheduled maintenance periods when few users are active on the system, you may want to configure a high value for sort buffers. To speed your index maintenance, you may want to benchmark performance of high sort buffer values, large I/O, and cache bindings to optimize your index activity.

- The reduction in merge runs is a logarithmic function. Increasing the value of number of sort buffers from 500 to 600 has very little effect on the number of merge runs. Increasing the size to a much larger value, such as 5000, can greatly speed the sort by reducing the number of merge runs and the amount of I/O needed.

- If number of sort buffers is set to less than the square root of the worktable size, sort performance is degraded. Since worktables include only columns specified in the select list plus columns needed for later joins, worktable size for merge joins is usually considerably smaller than the original table size.

When enough sort buffers are configured, fewer intermediate steps and merge runs need to take place during a sort, and physical I/O is required. When number of sort buffers is equal to or greater than the number of pages in the table, the sort can be performed completely in cache, with no physical I/O for the intermediate steps: the only I/O required is the I/O to read and write the data and index pages.

## Using less than the configured number of sort buffers

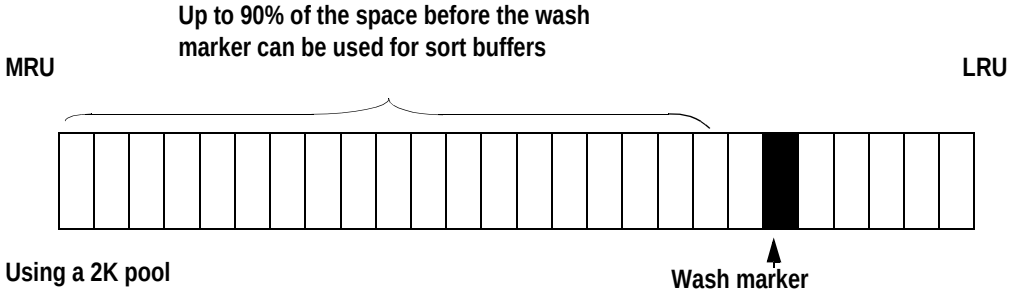
There are two types of sorts that may use fewer than the configured number of sort buffers:

- Creating a clustered index on a partition table always requires a parallel sort. If the table size is smaller than the number of configured sort buffers, then the sort reserves the number of pages in the table for the sort.
- Small serial sorts reserve just the number of sort buffers required to hold the table in cache.

## Configuring the *number of sort buffers* parameter

When creating indexes in parallel, the number of sort buffers must be equal to or less than 90 percent of the number of buffers in the pool area, before the wash marker, as shown in Figure 24-2.

***Figure 24-2: Area available for sort buffers***



The limit of 90 percent of the pool size is not enforced when you configure the number of sort buffers parameter, but it is enforced when you run the create index command, since the limit is enforced on the pool for the table being sorted. The maximum value that can be set for number of sort buffers is 32,767; this value is enforced by sp_configure.

#### Computing the allowed sort buffer value for a pool

sp_cacheconfig returns the size of the pool in megabytes and the wash size in kilobytes. For example, this output shows the size of the pools in the default data cache:

```
Cache: default data cache,   Status: Active,   Type: Default
      Config Size: 0.00 Mb,   Run Size: 38.23 Mb
      Config Replacement: strict LRU,   Run Replacement: strict LRU
      Config Partition:            2,   Run Partition:            2
 IO Size  Wash Size Config Size  Run Size     APF Percent
 -------- --------- ------------ ------------ -----------
     2 Kb   4544 Kb      0.00 Mb     22.23 Mb     10
    16 Kb   3200 Kb     16.00 Mb     16.00 Mb     10
```

This procedure takes the size of the 2K pool and its wash size as parameters, converts both values to pages and computes the maximum number of pages that can be used for sort buffers:

```
create proc bufs @poolsize numeric(6,2), @wash int
as
select "90% of non-wash 2k pool" =
    ((@poolsize * 512) - (@wash/2)) * .9
```

The following example executes bufs with values of “22.23 Mb” for the pool size and “4544 Kb” for the wash size:

```
bufs 22.23, 4544
```

The bufs procedure returns the following results:

```
90% of non-wash 2k pool
-----------------------
               8198.784
```

This command sets the number of sort buffers to 8198 pages:

```
sp_configure "number of sort buffers", 8198
```

If the table on which you want to create the index is bound to a user-defined cache, configure the appropriate number of sort buffers for the specific cache. As an alternative, you can unbind the table from the cache, create the index, and rebind the table:

```
sp_unbindcache pubtune, titles
create clustered index title_ix
    on titles (title_id)
sp_bindcache pubtune_cache, pubtune, titles
```

**Warning!** The buffers used by a sort are reserved entirely for the use of the sort until the sort completes. They cannot be used by another other task on the server. Setting the number of sort buffers to 90 percent of the pool size can seriously affect query processing if you are creating indexes while other transactions are active.

## Procedure for estimating merge levels and I/O

The following procedure estimates the number of merge runs and the amount of physical I/O required to create an index:

```
create proc merge_runs @pages int, @bufs int
as
declare @runs int, @merges int, @maxmerge int

select @runs = ceiling ( @pages / @bufs )

/* if all pages fit into sort buffers, no merge runs needed */
if @runs <=1
        select @merges = 0
else
begin
    if @runs > @bufs select @maxmerge = @bufs
    else  select @maxmerge = @runs

    if @maxmerge < 2 select @maxmerge = 2

    select @merges = ceiling(log10(@runs) / log10(@maxmerge))
end
select @merges "Merge Levels",
        2 * @pages * @merges + @pages "Total IO"
```

The parameters for the procedure are:

- *pages* – the number of pages in the table, or the number of leaf-level pages in a nonclustered index.
- *bufs* – the number of sort buffers to configure.

This example uses the default number of sort buffers for a table with 2,000,000 pages:

```
merge_runs 2000000, 500, 20
```

The merge_runs procedure estimates that 2 merge runs and 10,000,000 I/Os would be required to create the index:

```
 Merge Levels Total IO
 ------------ -----------
            2    10000000
```

Increasing the number of sort buffers to 1500 reduces the number of merge runs and the I/O required:

```
merge_runs 2000000, 1500
 Merge Levels Total IO
 ------------ -----------
            1     6000000
```

The total I/O predicted by this procedure may be different than the I/O usage on your system, depending on the size and configuration of the cache and pools used by the sort.

## Configuring caches for large I/O during parallel sorting

Sorts can use large I/O:

- During the sampling phase
- For the producers scanning the input tables
- For the consumers performing disk I/O on intermediate and final sort results

For these steps, sorts can use the largest pool size available in the cache used by the table being sorted; they can use the 2K pool if no large I/O buffers are available.

### Balancing sort buffers and large I/O configuration

Configuring a pool for 16K buffers in the cache used by the sort greatly speeds I/O for the sort, substantially reducing the number of physical I/Os for a sort. Part of this I/O savings results from using large I/O to scan the input table.

Additional I/O, both reads and writes, takes place during merge phases of the sort. The amount of I/O during this step depends on the number of merge phases required. During the sort and merge step, buffers are either read once and not needed again, or they are filled with intermediate sort output results, written to disk, and available for reuse. The cache-hit ratio during sorts will always be low, so configuring a large 16K cache wastes space that can better be used for sort buffers, to reduce merge runs.

For example, creating a clustered index on a 250MB table using a 32MB cache performed optimally with only 4MB configured in the 16K pool and 10,000 sort buffers. Larger pool sizes did not affect the cache hit ratio or number of I/Os. Changing the wash size for the 16K pool to the maximum allowed helped performance slightly, since the small pool size tended to allow buffers to reach the LRU end of the cache before the writes were completed. The following formula computes the maximum allowable wash size for a 16K pool:

```
select floor((size_in_MB * 1024 /16) * .8) * 16
```

## Disk requirements

Disk requirements for parallel sorting are as follows:

- Space is needed to store the completed index.
- Having multiple devices in the target segment increases the number of consumers for worktable sorts and for creating nonclustered indexes and clustered indexes on non partitioned tables.

## Space requirements for creating indexes

Creating indexes requires space to store the sorted index. For clustered indexes, this requires copying the data rows to new locations in the order of the index key. The newly ordered data rows and the upper levels of the index must be written before the base table can be removed. Unless you are using the with sorted_data clause to suppress the sort, creating a clustered index requires approximately 120 percent of the space occupied by the table.

Creating a nonclustered index requires space to store the new index. To help determine the size of objects and the space that is available, use the following system procedures:

- sp_spaceused – to see the size of the table. See "Using sp_spaceused to Display Object Size" on page 338.
- sp_estspace – to predict the size of the index. See "Using sp_estspace to Estimate Object Size" on page 340.
- sp_helpsegment – to see space left on a database segment. See "Checking data distribution on devices with sp_helpsegment" on page 101.

## Space requirements for worktable sorts

Queries that sort worktables (merge joins and order by, distinct, union, and reformatting) first copy the needed columns for the query into the worktable and then perform the sort. These worktables are stored on the system segment in tempdb, so this is the target segment for queries that require sorts. To see the space available and the number of devices, use:

```
tempdb..sp_helpsegment system
```

The process of inserting the rows into the worktable and the parallel sort do not require multiple devices to operate in parallel. However, performance improves when the system segment in tempdb spans multiple database devices.

## Number of devices in the target segment

As described in "Worker process requirements for parallel sorts" on page 581, the number of devices in the target segment determines the number of consumers for sort operations, except for creating a clustered index on a partitioned table.

Performance considerations for query processing, such as the improvements in I/O when indexes are on separate devices from the data are more important in determining your device allocations and object placement than sort requirements.

If your worktable sorts are large enough to require parallel sorts, multiple devices in the system segment of tempdb will speed these sorts, as well as increase I/O parallelism while rows are being inserted into the worktable.

# Recovery considerations

Creating indexes is a minimally-logged database operation. Serial sorts are recovered from the transaction log by completely redoing the sort. However, parallel create index commands are not recoverable from the transaction log—after performing a parallel sort, you must dump the database before you can use the dump transaction command on the database.

Adaptive Server does not automatically perform parallel sorting for create index commands unless the select into/bulk copy/pllsort database option is set on. Creating a clustered index on a partitioned table always requires a parallel sort; other sort operations can be performed in serial if the select into/bulk copy/pllsort option is not enabled.

# Tools for observing and tuning sort behavior

Adaptive Server provides several tools for working with sort behavior:

- set sort_resources on shows how a create index command would be performed, without creating the index. See “Using set sort_resources on” on page 595.
- Several system procedures can help estimate the size, space, and time requirements:
  - sp_configure – Displays configuration parameters. See “Configuration parameters for controlling parallelism” on page 517.

- sp_helpartition – Displays information about partitioned tables. See “Getting information about partitions” on page 98.
- sp_helpsegment – Displays information about segments, devices, and space usage. See “Checking data distribution on devices with sp_helpsegment” on page 101.
- sp_sysmon – Reports on many system resources used for parallel sorts, including CPU utilization, physical I/O, and caching. See “Using sp_sysmon to tune index creation” on page 599.

## Using *set sort_resources on*

The set sort_resources on command can help you understand how the sort manager performs parallel sorting for create index statements. You can use it before creating an index to determine whether you want to increase configuration parameters or specify additional consumers for a sort.

After you use set sort_resources on, Adaptive Server does not actually create indexes, but analyzes resources, performs the sampling step, and prints detailed information about how Adaptive Server would use parallel sorting to execute the create index command. Table 24-2 describes the messages that can be printed for sort operations.

***Table 24-2: Basic sort resource messages***

| Message | Explanation | See |
|---|---|---|
| `The Create Index is done using` *`sort_type`* | *sort_type* is either “Parallel Sort” or “Serial Sort.” | “Requirements and resources overview” on page 576 |
| `Sort buffer size:` *`N`* | *N* is the configured value for the number of sort buffers configuration parameter. | “Sort buffer configuration guidelines” on page 587 |
| `Parallel degree:` *`N`* | *N* is the maximum number of worker processes that the parallel sort can use, as set by configuration parameters. | “Caches, sort buffers, and parallel sorts” on page 585 |
| `Number of output devices:` *`N`* | *N* is the total number of database devices on the target segment. | “Disk requirements” on page 592 |
| `Number of producer threads:` *`N`* | *N* is the optimal number of producer processes determined by the sort manager. | “Worker process requirements for parallel sorts” on page 581 |
| `Number of consumer threads:` *`N`* | *N* is the optimal number of consumer processes determined by the sort manager. | “Worker process requirements for parallel sorts” on page 581 |

| Message | Explanation | See |
|---|---|---|
| `The distribution map contains M element(s) for N partitions.` | *M* is the number of elements that define range boundaries in the distribution map. *N* is the total number of partitions (ranges) in the distribution map. | "Creating a distribution map" on page 579 |
| `Partition Element:N value` | *N* is the number of the distribution map element. *value* is the distribution map element that defines the boundary of each partition. | "Creating a distribution map" on page 579 |
| `Number of sampled records: N` | *N* is the number of sampled records used to create the distribution map. | "Creating a distribution map" on page 579 |

## Examples

The following examples show the output of the set sort_resources command.

### Nonclustered index on a nonpartitioned table

This example shows how Adaptive Server performs parallel sorting for a create index command on an unpartitioned table. Pertinent details for the example are:

- The default segment spans 4 database devices.
- max parallel degree is set to 20 worker processes.
- number of sort buffers is set to the default, 500 buffers.

The following commands set sort_resources on and issue a create index command on the orders table:

```
set sort_resources on
create index order_ix on orders (order_id)
```

Adaptive Server prints the following output:

```
The Create Index is done using Parallel Sort
Sort buffer size: 500
Parallel degree: 20
Number of output devices: 4
Number of producer threads: 1
Number of consumer threads: 4
The distribution map contains 3 element(s) for 4
partitions.
Partition Element: 1
```

```
458052

Partition Element: 2

909063

Partition Element: 3

1355747

Number of sampled records: 2418
```

In this example, the 4 devices on the default segment determine the number of consumer processes for the sort. Because the input table is not partitioned, the sort manager allocates 1 producer process, for a total degree of parallelism of 5.

The distribution map uses 3 dividing values for the 4 ranges. The lowest input values up to and including the value 458052 belong to the first range. Values greater than 458052 and less than or equal to 909063 belong to the second range. Values greater than 909063 and less than or equal to 1355747 belong to the third range. Values greater than 1355747 belong to the fourth range.

#### Nonclustered index on a partitioned table

This example uses the same tables and devices as the first example. However, in this example, the input table is partitioned before creating the nonclustered index. The commands are:

```
set sort_resources on
alter table orders partition 9
create index order_ix on orders (order_id)
```

In this case, the create index command under the sort_resources option prints the output:

```
The Create Index is done using Parallel Sort
Sort buffer size: 500
Parallel degree: 20
Number of output devices: 4
Number of producer threads: 9
Number of consumer threads: 4
The distribution map contains 3 element(s) for 4
partitions.
Partition Element: 1
```

```
458464
Partition Element: 2

892035
Partition Element: 3

1349187
Number of sampled records: 2448
```

Because the input table is now partitioned, the sort manager allocates 9 producer threads, for a total of 13 worker processes. The number of elements in the distribution map is the same, although the values differ slightly from those in the previous sort examples.

#### Clustered index on partitioned table executed in parallel

This example creates a clustered index on orders, specifying the segment name, order_seg.

```
set sort_resources on
alter table orders partition 9
create clustered index order_ix
    on orders (order_id) on order_seg
```

Since the number of available worker processes is 20, this command can use 9 producers and 9 consumers, as shown in the output:

```
The Create Index is done using Parallel Sort
Sort buffer size: 500
Parallel degree: 20
Number of output devices: 9
Number of producer threads: 9
Number of consumer threads: 9
The distribution map contains 8 element(s) for 9
partitions.
Partition Element: 1

199141
Partition Element: 2

397543
Partition Element: 3

598758
Partition Element: 4

800484
```

```
Partition Element: 5

1010982
Partition Element: 6

1202471
Partition Element: 7

1397664
Partition Element: 8

1594563
Number of sampled records: 8055
```

This distribution map contains 8 elements for the 9 partitions on the table being sorted. The number of worker processes used is 18.

#### Sort failure

For example, if only 10 worker processes had been available for this command, it could have succeeded using a single producer process to read the entire table. If fewer than 10 worker processes had been available, a warning message would be printed instead of the sort_resources output:

```
Msg 1538, Level 17, State 1:
Server 'snipe', Line 1:
Parallel degree 8 is less than required parallel
degree 10 to create clustered index on partition
table. Change the parallel degree to required
parallel degree and retry.
```

## Using *sp_sysmon* to tune index creation

You can use the "begin_sample" and "end_sample" syntax for sp_sysmon to provide performance results for individual create index commands:

```
sp_sysmon begin_sample
create index ...
sp_sysmon end_sample
```

Sections of the report to check include:

- The "Sample Interval," for the total time taken to create the index
- Cache statistics for the cache used by the table

    - Check the value for “Buffer Grabs” for the 2K and 16K pools to determine the effectiveness of large I/O.
    - Check the value “Dirty Buffer Grabs,” If this value is nonzero, set the wash size in the pool higher and/or increase the pool size, using sp_poolconfig.
- Disk I/O for the disks used by the table and indexes: check the value for “Total Requested I/Os”

CHAPTER 25

# Tuning Asynchronous Prefetch

This chapter explains how asynchronous prefetch improves I/O performance for many types of queries by reading data and index pages into cache before they are needed by the query.



## How asynchronous prefetch improves performance

Asynchronous prefetch improves performance by anticipating the pages required for certain well-defined classes of database activities whose access patterns are predictable. The I/O requests for these pages are issued before the query needs them so that most pages are in cache by the time query processing needs to access the page. Asynchronous prefetch can improve performance for:

- Sequential scans, such as table scans, clustered index scans, and covered nonclustered index scans
- Access via nonclustered indexes
- Some dbcc checks and update statistics
- Recovery

Asynchronous prefetch can improve the performance of queries that access large numbers of pages, such as decision support applications, as long as the I/O subsystems on the machine are not saturated.

Asynchronous prefetch cannot help (or may help only slightly) when the I/O subsystem is already saturated or when Adaptive Server is CPU-bound. It may be used in some OLTP applications, but to a much lesser degree, since OLTP queries generally perform fewer I/O operations.

When a query in Adaptive Server needs to perform a table scan, it:

- Examines the rows on a page and the values in the rows.
- Checks the cache for the next page to be read from a table. If that page is in cache, the task continues processing. If the page is not in cache, the task issues an I/O request and sleeps until the I/O completes.
- When the I/O completes, the task moves from the sleep queue to the run queue. When the task is scheduled on an engine, Adaptive Server examines rows on the newly fetched page.

This cycle of executing and stalling for disk reads continues until the table scan completes. In a similar way, queries that use a nonclustered index process a data page, issue the I/O for the next page referenced by the index, and sleep until the I/O completes, if the page is not in cache.

This pattern of executing and then waiting for I/O slows performance for queries that issue physical I/Os for large number of pages. In addition to the waiting time for the physical I/Os to complete, the task switches on and off the engine repeatedly. This task switching adds overhead to processing.

## Improving query performance by prefetching pages

Asynchronous prefetch issues I/O requests for pages before the query needs them so that most pages are in cache by the time query processing needs to access the page. If required pages are already in cache, the query does not yield the engine to wait for the physical read. (It may still yield for other reasons, but it yields less frequently.)

Based on the type of query being executed, asynchronous prefetch builds a **look-ahead set** of pages that it predicts will be needed very soon. Adaptive Server defines different look-ahead sets for each processing type where asynchronous prefetch is used.

In some cases, look-ahead sets are extremely precise; in others, some assumptions and speculation may lead to pages being fetched that are never read. When only a small percentage of unneeded pages are read into cache, the performance gains of asynchronous prefetch far outweigh the penalty for the wasted reads. If the number of unused pages becomes large, Adaptive Server detects this condition and either reduces the size of the look-ahead set or temporarily disables prefetching.

## Prefetching control mechanisms in a multiuser environment

When many simultaneous queries are prefetching large numbers of pages into a buffer pool, there is a risk that the buffers fetched for one query could be flushed from the pool before they are used.

Adaptive Server tracks the buffers brought into each pool by asynchronous prefetch and the number that are used. It maintains a per-pool count of prefetched but unused buffers. By default, Adaptive Server sets an asynchronous prefetch limit of 10 percent of each pool. In addition, the limit on the number of prefetched but unused buffers is configurable on a per-pool basis.

The pool limits and usage statistics act like a governor on asynchronous prefetch to keep the cache-hit ratio high and reduce unneeded I/O. Overall, the effect is to ensure that most queries experience a high cache-hit ratio and few stalls due to disk I/O sleeps.

The following sections describe how the look-ahead set is constructed for the activities and query types that use asynchronous prefetch. In some asynchronous prefetch optimizations, allocation pages are used to build the look-ahead set.

For information on how allocation pages record information about object storage, see “Allocation pages” on page 146.

## Look-ahead set during recovery

During recovery, Adaptive Server reads each log page that includes records for a transaction and then reads all the data and index pages referenced by that transaction, to verify timestamps and to roll transactions back or forward. Then, it performs the same work for the next completed transaction, until all transactions for a database have been processed. Two separate asynchronous prefetch activities speed recovery: asynchronous prefetch on the log pages themselves and asynchronous prefetch on the referenced data and index pages.

### Prefetching log pages

The transaction log is stored sequentially on disk, filling extents in each allocation unit. Each time the recovery process reads a log page from a new allocation unit, it prefetches all the pages on that allocation unit that are in use by the log.

In databases that do not have a separate log segment, log and data extents may be mixed on the same allocation unit. Asynchronous prefetch still fetches all the log pages on the allocation unit, but the look-ahead sets may be smaller.

### Prefetching data and index pages

For each transaction, Adaptive Server scans the log, building the look-ahead set from each referenced data and index page. While one transaction's log records are being processed, asynchronous prefetch issues requests for the data and index pages referenced by subsequent transactions in the log, reading the pages for transactions ahead of the current transaction.

---

**Note** Recovery uses only the pool in the default data cache. See "Setting limits for recovery" on page 615 for more information.

---

## Look-ahead set during sequential scans

Sequential scans include table scans, clustered index scans, and covered nonclustered index scans.

During table scans and clustered index scans, asynchronous prefetch uses allocation page information about the pages used by the object to construct the look-ahead set. Each time a page is fetched from a new allocation unit, the look-ahead set is built from all the pages on that allocation unit that are used by the object.

The number of times a sequential scan hops between allocation units is kept to measure fragmentation of the page chain. This value is used to adapt the size of the look-ahead set so that large numbers of pages are prefetched when fragmentation is low, and smaller numbers of pages are fetched when fragmentation is high. For more information, see "Page chain fragmentation" on page 609.

## Look-ahead set during nonclustered index access

When using a nonclustered index to access rows, asynchronous prefetch finds the page numbers for all qualified index values on a nonclustered index leaf page. It builds the look-ahead set from the unique list of all the pages that are needed.

Asynchronous prefetch is used only if two or more rows qualify.

If a nonclustered index access requires several leaf-level pages, asynchronous prefetch requests are also issued on the leaf pages.

## Look-ahead set during *dbcc* checks

Asynchronous prefetch is used during the following dbcc checks:

- dbcc checkalloc, which checks allocation for all tables and indexes in a database, and the corresponding object-level commands, dbcc tablealloc and dbcc indexalloc
- dbcc checkdb, which checks all tables and index links in a database, and dbcc checktable, which checks individual tables and their indexes

### Allocation checking

The dbcc commands checkalloc, tablealloc and indexalloc, which check page allocations validate information on the allocation page. The look-ahead set for the dbcc operations that check allocation is similar to the look-ahead set for other sequential scans. When the scan enters a different allocation unit for the object, the look-ahead set is built from all the pages on the allocation unit that are used by the object.

### *checkdb* and *checktable*

The dbcc checkdb and dbcc checktable commands check the page chains for a table, building the look-ahead set in the same way as other sequential scans.

If the table being checked has nonclustered indexes, they are scanned recursively, starting at the root page and following all pointers to the data pages. When checking the pointers from the leaf pages to the data pages, the dbcc commands use asynchronous prefetch in a way that is similar to nonclustered index scans. When a leaf-level index page is accessed, the look-ahead set is built from the page IDs of all the pages referenced on the leaf-level index page.

## Look-ahead set minimum and maximum sizes

The size of a look-ahead set for a query at a given point in time is determined by several factors:

- The type of query, such as a sequential scan or a nonclustered index scan
- The size of the pools used by the objects that are referenced by the query and the prefetch limit set on each pool
- The fragmentation of tables or indexes, in the case of operations that perform scans
- The recent success rate of asynchronous prefetch requests and overload conditions on I/O queues and server I/O limits

Table 25-1 summarizes the minimum and maximum sizes for different type of asynchronous prefetch usage.

***Table 25-1: Look-ahead set sizes***

| Access type | Action | Look-ahead set sizes |
|---|---|---|
| Table scan<br>Clustered index scan<br>Covered leaf level scan | Reading a page from a new allocation unit | Minimum is 8 pages needed by the query<br>Maximum is the smaller of:<br>• The number of pages on an allocation unit that belong to an object (at 2K, maximum is 255; 256 minus the allocation page).<br>• The pool prefetch limits |
| Nonclustered index scan | Locating qualified rows on the leaf page and preparing to access data pages | Minimum is 2 qualified rows<br>Maximum is the smaller of:<br>• The number of unique page numbers on qualified rows on the leaf index page<br>• The pool's prefetch limit |
| Recovery | Recovering a transaction | Maximum is the smaller of:<br>• All of the data and index pages touched by a transaction undergoing recovery<br>• The prefetch limit of the pool in the default data cache |
| | Scanning the transaction log | Maximum is all pages on an allocation unit belonging to the log |
| dbcc tablealloc, indexalloc, and checkalloc | Scanning the page chain | Same as table scan |
| dbcc checktable and checkdb | Scanning the page chain | Same as table scan |
| | Checking nonclustered index links to data pages | All of the data pages referenced on a leaf level page. |

# When prefetch is automatically disabled

Asynchronous prefetch attempts to fetch needed pages into buffer pools without flooding the pools or the I/O subsystem and without reading unneeded pages. If Adaptive Server detects that prefetched pages are being read into cache but not used, it temporarily limits or discontinues asynchronous prefetch.

## Flooding pools

For each pool in the data caches, a configurable percentage of buffers can be read in by asynchronous prefetch and held until their first use. For example, if a 2K pool has 4000 buffers, and the limit for the pool is 10 percent, then, at most, 400 buffers can be read in by asynchronous prefetch and remain unused in the pool. If the number of nonaccessed prefetched buffers in the pool reaches 400, Adaptive Server temporarily discontinues asynchronous prefetch for that pool.

As the pages in the pool are accessed by queries, the count of unused buffers in the pool drops, and asynchronous prefetch resumes operation. If the number of available buffers is smaller than the number of buffers in the look-ahead set, only that many asynchronous prefetches are issued. For example, if 350 unused buffers are in a pool that allows 400, and a query's look-ahead set is 100 pages, only the first 50 asynchronous prefetches are issued.

This keeps multiple asynchronous prefetch requests from flooding the pool with requests that flush pages out of cache before they can be read. The number of asynchronous I/Os that cannot be issued due to the per-pool limits is reported by sp_sysmon.

## I/O system overloads

Adaptive Server and the operating system place limits on the number of outstanding I/Os for the server as a whole and for each engine. The configuration parameters max async i/os per server and max async i/os per engine control these limits for Adaptive Server. See your operating system documentation for more information on configuring them for your hardware.

The configuration parameter disk i/o structures controls the number of disk control blocks that Adaptive Server reserves. Each physical I/O (each buffer read or written) requires one control block while it is in the I/O queue.

See the *System Administration Guide*.

If Adaptive Server tries to issue asynchronous prefetch requests that would exceed max async i/os per server, max async i/os per engine, or disk i/o structures, it issues enough requests to reach the limit and discards the remaining requests. For example, if only 50 disk I/O structures are available, and the server attempts to prefetch 80 pages, 50 requests are issued, and the other 30 are discarded.

sp_sysmon reports the number of times these limits are exceeded by asynchronous prefetch requests. See "Asynchronous prefetch activity report" on page 978.

## Unnecessary reads

Asynchronous prefetch tries to avoid unnecessary physical reads. During recovery and during nonclustered index scans, look-ahead sets are exact, fetching only the pages referenced by page number in the transaction log or on index pages.

Look-ahead sets for table scans, clustered index scans, and dbcc checks are more speculative and may lead to unnecessary reads. During sequential scans, unnecessary I/O can take place due to:

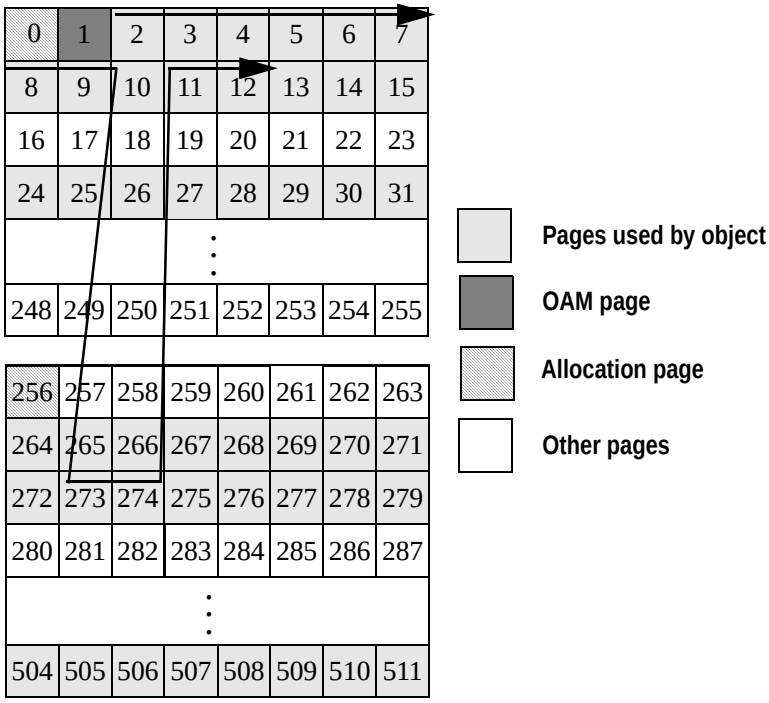
- Page chain fragmentation on allpages-locked tables
- Heavy cache utilization by multiple users

### Page chain fragmentation

Adaptive Server's page allocation mechanism strives to keep pages that belong to the same object close to each other in physical storage by allocating new pages on an extent already allocated to the object and by allocating new extents on allocation units already used by the object.

However, as pages are allocated and deallocated, page chains on data-only-locked tables can develop kinks. Figure 25-1 shows an example of a kinked page chain between extents in two allocation units.

***Figure 25-1: A kink in a page chain crossing allocation units***




In Figure 25-1, when a scan first needs to access a page from allocation unit 0, it checks the allocation page and issues asynchronous I/Os for all the pages used by the object it is scanning, up to the limit set on the pool. As the pages become available in cache, the query processes them in order by following the page chain. When the scan reaches page 10, the next page in the page chain, page 273, belongs to allocation unit 256.

When page 273 is needed, allocation page 256 is checked, and asynchronous prefetch requests are issued for all the pages in that allocation unit that belong to the object.

When the page chain points back to a page in allocation unit 0, there are two possibilities:

- The prefetched pages from allocation unit 0 are still in cache, and the query continues processing with no unneeded physical I/Os.

- The prefetch pages from allocation unit 0 have been flushed from the cache by the reads from allocation unit 256 and other I/Os taking place by other queries that use the pool. The query must reissue the prefetch requests. This condition is detected in two ways:
  - Adaptive Server's count of the hops between allocation pages now equals two. It uses the ratio between the count of hops and the prefetched pages to reduce the size of the look-ahead set, so fewer I/Os are issued.
  - The count of prefetched but unused pages in the pool is likely to be high, so asynchronous prefetch may be temporarily discontinued or reduced, based on the pool's limit.

# Tuning Goals for Asynchronous Prefetch

Choosing optimal pool sizes and prefetch percentages for buffer pools can be key to achieving improved performance with asynchronous prefetch. When multiple applications are running concurrently, a well-tuned prefetching system balances pool sizes and prefetch limits to accomplish these goals:

- Improved system throughput
- Better performance by applications that use asynchronous prefetch
- No performance degradation in applications that do not use asynchronous prefetch

Configuration changes to pool sizes and the prefetch limits for pools are dynamic, allowing you to make changes to meet the needs of varying workloads. For example, you can configure asynchronous prefetch for good performance during recovery or `dbcc` checking and reconfigure afterward without needing to restart Adaptive Server.

See "Setting limits for recovery" on page 615 and "Setting limits for dbcc" on page 616 for more information.

## Commands for configuration

Asynchronous prefetch limits are configured as a percentage of the pool in which prefetched but unused pages can be stored. There are two configuration levels:

- The server-wide default, set with the configuration parameter global async prefetch limit. When you first install, the default value for global async prefetch limit is 10 (percent).

  For more information, see of the *System Administration Guide*.

- A per-pool override, set with sp_poolconfig. To see the limits set for each pool, use sp_cacheconfig.

  For more information, see of the *System Administration Guide*.

Changing asynchronous prefetch limits takes effect immediately, and does not require a reboot. Both the global and per-pool limits can also be configured in the configuration file.

# Other Adaptive Server performance features

This section covers the interaction of asynchronous prefetch with other Adaptive Server performance features.

## Large I/O

The combination of large I/O and asynchronous prefetch can provide rapid query processing with low I/O overhead for queries performing table scans and for dbcc operations.

When large I/O prefetches all the pages on an allocation unit, the minimum number of I/Os for the entire allocation unit is:

- 31 16K I/Os
- 7 2K I/Os, for the pages that share an extent with the allocation page

## Sizing and limits for the 16k pool

Performing 31 16K prefetches with the default asynchronous prefetch limit of 10 percent of the buffers in the pool requires a pool with at least 310 16K buffers. If the pool is smaller, or if the limit is lower, some prefetch requests will be denied. To allow more asynchronous prefetch activity in the pool, you can configure a larger pool or a larger prefetch limit for the pool.

If multiple overlapping queries perform table scans using the same pool, the number of unused, prefetched pages allowed in the poll needs to be higher. The queries are probably issuing prefetch requests at slightly staggered times and are at different stages in reading the accessed pages. For example, one query may have just prefetched 31 pages, and have 31 unused pages in the pool, while an earlier query has only 2 or 3 unused pages left. To start your tuning efforts for these queries, assume one-half the number of pages for a prefetch request multiplied by the number of active queries in the pool.

## Limits for the 2K pool

Queries using large I/O during sequential scans may still need to perform 2K I/O:

- When a scan enters a new allocation unit, it performs 2K I/O on the 7 pages in the unit that share space with the allocation page.
- If pages from the allocation unit already reside in the 2K pool when the prefetch requests are issued, the pages that share that extent must be read into the 2K pool.

If the 2K pool has its asynchronous prefetch limit set to 0, the first 7 reads are performed by normal asynchronous I/O, and the query sleeps on each read if the pages are not in cache. Set the limits on the 2K pool high enough that it does not slow prefetching performance.

## Fetch-and-discard (MRU) scans

When a scan uses MRU replacement policy, buffers are handled in a special manner when they are read into the cache by asynchronous prefetch. First, pages are linked at the MRU end of the chain, rather than at the wash marker. When the query accesses the page, the buffers are re linked into the pool at the wash marker. This strategy helps to avoid cases where heavy use of a cache flushes prefetched buffers linked at the wash marker before they can be used. It has little impact on performance, unless large numbers of unneeded pages are being prefetched. In this case, the prefetched pages are more likely to flush other pages from cache.

## Parallel scans and large I/Os

The demand on buffer pools can become higher with parallel queries. With serial queries operating on the same pools, it is safe to assume that queries are issued at slightly different times and that the queries are in different stages of execution: some are accessing pages are already in cache, and others are waiting on I/O.

Parallel execution places different demands on buffer pools, depending on the type of scan and the degree of parallelism. Some parallel queries are likely to issue a large number of prefetch requests simultaneously.

### Hash-based table scans

Hash-based table scans on allpages-locked tables have multiple worker processes accessing the same page chain. Each worker process checks the page ID of each page in the table, but examines only the rows on those pages where page ID matches the hash value for the worker process.

The first worker process that needs a page from a new allocation unit issues a prefetch request for all pages from that unit. When the scans of other worker processes also need pages from that allocation unit, they will either find that the pages they need are already in I/O or already in cache. As the first scan to complete enters the next unit, the process is repeated.

As long as one worker process in the family performing a hash-based scan does not become stalled (waiting for a lock, for example), the hash-based table scans do not place higher demands on the pools than they place on serial processes. Since the multiple processes may read the pages much more quickly than a serial process does, they change the status of the pages from unused to used more quickly.

### Partition-based scans

Partition-based scans are more likely to create additional demands on pools, since multiple worker processes may be performing asynchronous prefetching on different allocation units. On partitioned tables on multiple devices, the per-server and per-engine I/O limits are less likely to be reached, but the per-pool limits are more likely to limit prefetching.

Once a parallel query is parsed and compiled, it launches its worker processes. If a table with 4 partitions is being scanned by 4 worker processes, each worker process attempts to prefetch all the pages in its first allocation unit. For the performance of this single query, the most desirable outcome is that the size and limits on the 16K pool are sufficiently large to allow 124 (31*4) asynchronous prefetch requests, so all of the requests succeed. Each of the worker processes scans the pages in cache quickly, moving onto new allocation units and issuing more prefetch requests for large numbers of pages.

## Special settings for asynchronous prefetch limits

You may want to change asynchronous prefetch configuration temporarily for specific purposes, including:

- Recovery
- dbcc operations that use asynchronous prefetch

### Setting limits for recovery

During recovery, Adaptive Server uses only the 2K pool of the default data cache. If you shut down the server using shutdown with nowait, or if the server goes down due to power failure or machine failure, the number of log records to be recovered may be quite large.

To speed recovery, you can edit the configuration file to do one or both of the following:

- Increase the size of the 2K pool in the default data cache by reducing the size of other pools in the cache
- Increase the prefetch limit for the 2K pool

Both of these configuration changes are dynamic, so you can use sp_poolconfig to restore the original values after recovery completes, without restarting Adaptive Server. The recovery process allows users to log into the server as soon as recovery of the master database is complete. Databases are recovered one at a time and users can begin using a particular database as soon as it is recovered. There may be some contention if recovery is still taking place on some databases, and user activity in the 2K pool of the default data cache is heavy.

### Setting limits for *dbcc*

If you are performing database consistency checking at a time when other activity on the server is low, configuring high asynchronous prefetch limits on the pools used by dbcc can speed consistency checking.

dbcc checkalloc can use special internal 16K buffers if there is no 16K pool in the cache for the appropriate database. If you have a 2K pool for a database, and no 16K pool, set the local prefetch limit to 0 for the pool while executing dbcc checkalloc. Use of the 2K pool instead of the 16K internal buffers may actually hurt performance.

## Maintenance activities for high prefetch performance

Page chains for all pages-locked tables and the leaf levels of indexes develop kinks as data modifications take place on the table. In general, newly created tables have few kinks. Tables where updates, deletes, and inserts that have caused page splits, new page allocations, and page deallocations are likely to have cross-allocation unit page chain kinks. If more than 10 to 20 percent of the original rows in a table have been modified, you should determine if kinked page chains are reducing asynchronous prefetch effectiveness. If you suspect that page chain kinks are reducing asynchronous prefetch performance, you may need to re-create indexes or reload tables to reduce kinks.

## Eliminating kinks in heap tables

For allpages-locked heaps, page allocation is generally sequential, unless pages are deallocated by deletes that remove all rows from a page. These pages may be reused when additional space is allocated to the object. You can create a clustered index (and drop it, if you want the table stored as a heap) or bulk copy the data out, truncate the table, and copy the data in again. Both activities compress the space used by the table and eliminate page-chain kinks.

## Eliminating kinks in clustered index tables

For clustered indexes, page splits and page deallocations can cause page chain kinks. Rebuilding clustered indexes does not necessarily eliminate all cross-allocation page linkages. Use fillfactor for clustered indexes where you expect growth, to reduce the number of kinks resulting from data modifications.

## Eliminating kinks in nonclustered indexes

If your query mix uses covered index scans, dropping and re-creating nonclustered indexes can improve asynchronous prefetch performance, once the leaf-level page chain becomes fragmented.

# Performance monitoring and asynchronous prefetch

The output of statistics io reports the number physical reads performed by asynchronous prefetch and the number of reads performed by normal asynchronous I/O. In addition, statistics io reports the number of times that a search for a page in cache was found by the asynchronous prefetch without holding the cache spinlock.

See “Reporting physical and logical I/O statistics” on page 763 for more information.

sp_sysmon report contains information on asynchronous prefetch in both the “Data Cache Management” section and the “Disk I/O Management” section.

If you are using sp_sysmon to evaluate asynchronous prefetch performance, you may see improvements in other performance areas, such as:

- Much higher cache hit ratios in the pools where asynchronous prefetch is effective
- A corresponding reduction in context switches due to cache misses, with voluntary yields increasing
- A possible reduction in lock contention. Tasks keep pages locked during the time it takes for perform I/O for the next page needed by the query. If this time is reduced because asynchronous prefetch increases cache hits, locks will be held for a shorter time.

See “Data cache management” on page 973 and “Disk I/O management” on page 994 for more information.

CHAPTER 26

# *tempdb* Performance Issues

This chapter discusses the performance issues associated with using the tempdb database. tempdb is used by Adaptive Server users. Anyone can create objects in tempdb. Many processes use it silently. It is a server-wide resource that is used primarily for internal sorts processing, creating worktables, reformatting, and for storing temporary tables and indexes created by users.

Many applications use stored procedures that create tables in tempdb to expedite complex joins or to perform other complex data analysis that is not easily performed in a single step.



## How management of *tempdb* affects performance

Good management of tempdb is critical to the overall performance of Adaptive Server. tempdb cannot be overlooked or left in a default state. It is the most dynamic database on many servers and should receive special attention.

If planned for in advance, most problems related to tempdb can be avoided. These are the kinds of things that can go wrong if tempdb is not sized or placed properly:

- tempdb fills up frequently, generating error messages to users, who must then resubmit their queries when space becomes available.
- Sorting is slow, and users do not understand why their queries have such uneven performance.
- User queries are temporarily locked from creating temporary tables because of locks on system tables.
- Heavy use of tempdb objects flushes other pages out of the data cache.

## Main solution areas for *tempdb* performance

These main areas can be addressed easily:

- Sizing tempdb correctly for all Adaptive Server activity
- Placing tempdb optimally to minimize contention
- Binding tempdb to its own data cache
- Minimizing the locking of resources within tempdb

# Types and uses of temporary tables

The use or misuse of user-defined temporary tables can greatly affect the overall performance of Adaptive Server and your applications.

Temporary tables can be quite useful, often reducing the work the server has to do. However, temporary tables can add to the size requirement of tempdb. Some temporary tables are truly temporary, and others are permanent.

tempdb is used for three types of tables:

- Truly temporary tables
- Regular user tables
- Worktables

## Truly temporary tables

You can create truly temporary tables by using "#" as the first character of the table name:

```
create table #temptable (...)
```

or:

```
select select_list
    into #temptable ...
```

Temporary tables:

- Exist only for the duration of the user session or for the scope of the procedure that creates them
- Cannot be shared between user connections
- Are automatically dropped at the end of the session or procedure (or can be dropped manually)

When you create indexes on temporary tables, the indexes are stored in tempdb:

```
create index tempix on #temptable(col1)
```

## Regular user tables

You can create regular user tables in tempdb by specifying the database name in the command that creates the table:

```
create table tempdb..temptable (...)
```

or:

```
select select_list
    into tempdb..temptable
```

Regular user tables in tempdb:

- Can persist across sessions
- Can be used by bulk copy operations
- Can be shared by granting permissions on them
- Must be explicitly dropped by the owner (otherwise, they are removed when Adaptive Server is restarted)

You can create indexes in tempdb on permanent temporary tables:

```
create index tempix on tempdb..temptable(col1)
```

## Worktables

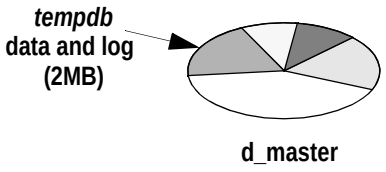
Worktables are automatically created in tempdb by Adaptive Server for merge joins, sorts, and other internal server processes. These tables:

- Are never shared
- Disappear as soon as the command completes

# Initial allocation of *tempdb*

When you install Adaptive Server, tempdb is 2MB, and is located completely on the master device, as shown in Figure 26-1. This is typically the first database that a System Administrator needs to make larger. The more users on the server, the larger it needs to be. It can be altered onto the master device or other devices. Depending on your needs, you may want to stripe tempdb across several devices.

***Figure 26-1: tempdb default allocation***




Use sp_helpdb to see the size and status of tempdb. The following example shows tempdb defaults at installation time:

```
                sp_helpdb tempdb
name      db_size  owner  dbid   created     status
--------- -------- ------ ------ ----------- --------------------
tempdb    2.0 MB   sa     2     May 22, 1999 select into/bulkcopy

device_frag  size    usage        free kbytes
------------ -------- ------------ ---------
master       2.0 MB  data and log 1248
```

# Sizing the *tempdb*

tempdb needs to be big enough to handle the following processes for every concurrent Adaptive Server user:

- Worktables for merge joins
- Worktables that are created for distinct, group by, and order by, for reformatting, and for the OR strategy, and for materializing some views and subqueries
- Temporary tables (those created with "#" as the first character of their names)
- Indexes on temporary tables
- Regular user tables in tempdb
- Procedures built by dynamic SQL

Some applications may perform better if you use temporary tables to split up multitable joins. This strategy is often used for:

- Cases where the optimizer does not choose a good query plan for a query that joins more than four tables
- Queries that join a very large number of tables
- Very complex queries
- Applications that need to filter data as an intermediate step

You might also use tempdb to:

- Denormalize several tables into a few temporary tables
- Normalize a denormalized table to do aggregate processing

For most applications, make tempdb 20 to 25% of the size of your user databases to provide enough space for these uses.

# Placing *tempdb*

Keep tempdb on separate physical disks from your critical application databases. Use the fastest disks available. If your platform supports solid state devices and your tempdb use is a bottleneck for your applications, use those devices. After you expand tempdb onto additional devices, drop the master device from the system, default, and logsegment segments.

Although you can expand tempdb on the same device as the master database,Sybase suggests that you use separate devices. Also, remember that logical devices, but not databases, are mirrored using Adaptive Server mirroring. If you mirror the master device, you create a mirror of all portions of the databases that reside on the master device. If the mirror uses serial writes, this can have a serious performance impact if your tempdb database is heavily used.

# Dropping the master Device from *tempdb* segments

By default, the system, default, and logsegment segments for tempdb include its 2MB allocation on the master device. When you allocate new devices to tempdb, they automatically become part of all three segments. Once you allocate a second device to tempdb, you can drop the master device from the default and logsegment segments. This way, you can be sure that the worktables and other temporary tables in tempdb do not contend with other uses on the master device.

To drop the master device from the segments:

1. Alter tempdb onto another device, if you have not already done so. For example:

   ```
   alter database tempdb on tune3 = 20
   ```

2. Issue a use tempdb command, and then drop the master device from the segments:

   ```
   sp_dropsegment "default", tempdb, master
   sp_dropdegment system, tempdb, master
   sp_dropdegment logsegment, tempdb, master
   ```

3. To verify that the default segment no longer includes the master device, issue this command:

```
select dbid, name, segmap
```

```
from sysusages, sysdevices
where sysdevices.low <= sysusages.size + vstart
  and sysdevices.high >= sysusages.size + vstart -1
  and dbid = 2
  and (status = 2 or status = 3)
```

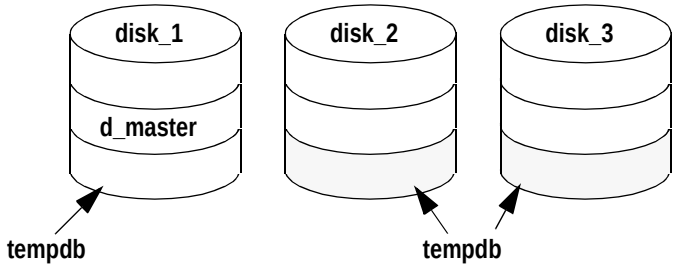
The segmap column should report "1" for any allocations on the master device, indicating that only the system segment still uses the device:

```
dbid   name            segmap
------ --------------- -----------
     2 master                    1
     2 tune3                     7
```

## Using multiple disks for parallel query performance

If tempdb spans multiple devices, as shown in Figure 26-2, you can take advantage of parallel query performance for some temporary tables or worktables.

***Figure 26-2: tempdb spanning disks***




# Binding *tempdb* to its own cache

Under normal Adaptive Server use, tempdb makes heavy use of the data cache as temporary tables are created, populated, and then dropped.

Assigning tempdb to its own data cache:

- Keeps the activity on temporary objects from flushing other objects out of the default data cache
- Helps spread I/O between multiple caches

See “Examining cache needs for tempdb” on page 320 for more information.

## Commands for cache binding

Use sp_cacheconfig and sp_poolconfig to create named data caches and to configure pools of a given size for large I/O. Only a System Administrator can configure caches and pools.

For instructions on configuring named caches and pools, see the *System Administration Guide*.

Once the caches have been configured, and the server has been restarted, you can bind tempdb to the new cache:

```
sp_bindcache "tempdb_cache", tempdb
```

# Temporary tables and locking

Creating or dropping temporary tables and their indexes can cause lock contention on the system tables in tempdb. When users create tables in tempdb, information about the tables must be stored in system tables such as sysobjects, syscolumns, and sysindexes. If multiple user processes are creating and dropping tables in tempdb, heavy contention can occur on the system tables. Worktables created internally do not store information in system tables.

If contention for tempdb system tables is a problem with applications that must repeatedly create and drop the same set of temporary tables, try creating the tables at the start of the application. Then use insert...select to populate them, and truncate table to remove all the data rows. Although insert...select requires logging and is slower than select into, it can provide a solution to the locking problem.

# Minimizing logging in *tempdb*

Even though the trunc log on checkpoint database option is turned on in tempdb, changes to tempdb are still written to the transaction log. You can reduce log activity in tempdb by:

- Using select into instead of create table and insert
- Selecting only the columns you need into the temporary tables

## With *select into*

When you create and populate temporary tables in tempdb, use the select into command, rather than create table and insert...select, whenever possible. The select into/bulkcopy database option is turned on by default in tempdb to enable this behavior.

select into operations are faster because they are only minimally logged. Only the allocation of data pages is tracked, not the actual changes for each data row. Each data insert in an insert...select query is fully logged, resulting in more overhead.

## By using shorter rows

If the application creating tables in tempdb uses only a few columns of a table, you can minimize the number and size of log records by:

- Selecting just the columns you need for the application, rather than using select * in queries that insert data into the tables
- Limiting the rows selected to just the rows that the applications requires

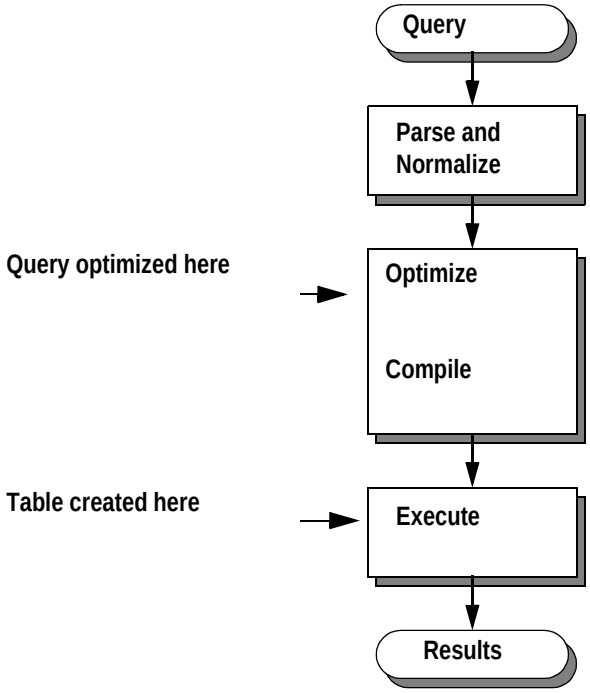
Both of these suggestions also keep the size of the tables themselves smaller.

# Optimizing temporary tables

Many uses of temporary tables are simple and brief and require little optimization. But if your applications require multiple accesses to tables in tempdb, you should examine them for possible optimization strategies. Usually, this involves splitting out the creation and indexing of the table from the access to it by using more than one procedure or batch.

When you create a table in the same stored procedure or batch where it is used, the query optimizer cannot determine how large the table is, the table has not yet been created when the query is optimized, as shown in Figure 26-3. This applies to both temporary tables and regular user tables.

***Figure 26-3: Optimizing and creating temporary tables***




The optimizer assumes that any such table has 10 data pages and 100 rows. If the table is really large, this assumption can lead the optimizer to choose a suboptimal query plan.

These two techniques can improve the optimization of temporary tables:

- Creating indexes on temporary tables

- Breaking complex use of temporary tables into multiple batches or procedures to provide information for the optimizer

## Creating indexes on temporary tables

You can define indexes on temporary tables. In many cases, these indexes can improve the performance of queries that use tempdb. The optimizer uses these indexes just like indexes on ordinary user tables. The only requirements are:

- The table must contain data when the index is created. If you create the temporary table and create the index on an empty table, Adaptive Server does not create column statistics such as histograms and densities. If you insert data rows after creating the index, the optimizer has incomplete statistics.
- The index must exist while the query using it is optimized. You cannot create an index and then use it in a query in the same batch or procedure.
- The optimizer may choose a suboptimal plan if rows have been added or deleted since the index was created or since update statistics was run.

Providing an index for the optimizer can greatly increase performance, especially in complex procedures that create temporary tables and then perform numerous operations on them.

## Creating nested procedures with temporary tables

You need to take an extra step to create the procedures described above. You cannot create base_proc until select_proc exists, and you cannot create select_proc until the temporary table exists. Here are the steps:

1 Create the temporary table outside the procedure. It can be empty; it just needs to exist and to have columns that are compatible with select_proc:

```
select * into #huge_result from ... where 1 = 2
```

2 Create the procedure select_proc, as shown above.

3 Drop #huge_result.

4 Create the procedure base_proc.

## Breaking *tempdb* uses into multiple procedures

For example, this query causes optimization problems with #huge_result:

```
create proc base_proc
as
    select *
        into #huge_result
        from ...
    select *
        from tab,
        #huge_result where ...
```

You can achieve better performance by using two procedures. When the base_proc procedure calls the select_proc procedure, the optimizer can determine the size of the table:

```
create proc select_proc
as
    select *
        from tab, #huge_result where ...
create proc base_proc
as
    select *
        into #huge_result
        from ...
    exec select_proc
```

If the processing for #huge_result requires multiple accesses, joins, or other processes, such as looping with while, creating an index on #huge_result may improve performance. Create the index in base_proc so that it is available when select_proc is optimized.

CHAPTER 27

# Cursors and Performance

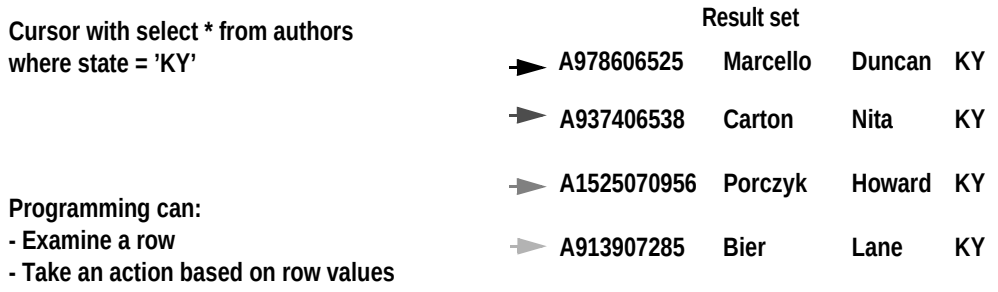
This chapter discusses performance issues related to cursors. Cursors are a mechanism for accessing the results of a SQL select statement one row at a time (or several rows, if you use set cursors rows). Since cursors use a different model from ordinary set-oriented SQL, the way cursors use memory and hold locks has performance implications for your applications. In particular, cursor performance issues includes locking at the page and at the table level, network resources, and overhead of processing instructions.



## Definition

A cursor is a symbolic name that is associated with a select statement. It enables you to access the results of a select statement one row at a time. Figure 27-1 shows a cursor accessing the authors table.

***Figure 27-1: Cursor example***




You can think of a cursor as a "handle" on the result set of a select statement. It enables you to examine and possibly manipulate one row at a time.

## Set-oriented versus row-oriented programming

SQL was conceived as a set-oriented language. Adaptive Server is extremely efficient when it works in set-oriented mode. Cursors are required by ANSI SQL standards; when they are needed, they are very powerful. However, they can have a negative effect on performance.

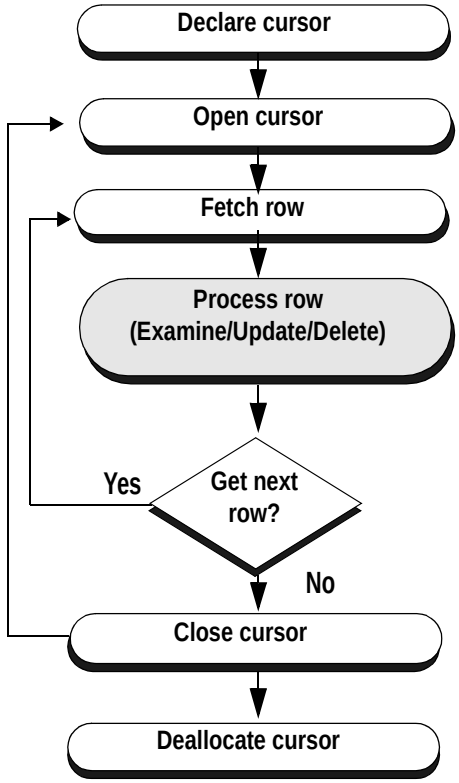
For example, this query performs the identical action on all rows that match the condition in the where clause:

```
update titles
    set contract = 1
where type = 'business'
```

The optimizer finds the most efficient way to perform the update. In contrast, a cursor would examine each row and perform single-row updates if the conditions were met. The application declares a cursor for a select statement, opens the cursor, fetches a row, processes it, goes to the next row, and so forth. The application may perform quite different operations depending on the values in the current row, and the server's overall use of resources for the cursor application may be less efficient than the server's set level operations. However, cursors can provide more flexibility than set-oriented programming.

Figure 27-2 shows the steps involved in using cursors. The function of cursors is to get to the middle box, where the user or application code examines a row and decides what to do, based on its values.

***Figure 27-2: Cursor flowchart***




## Example

Here is a simple example of a cursor with the "Process Rows" step shown above in pseudocode:

```
declare biz_book cursor
    for select * from titles
    where type = 'business'
go
open biz_book
go
fetch biz_book
go
/* Look at each row in turn and perform
** various tasks based on values,
```

```
** and repeat fetches, until
** there are no more rows
*/
close biz_book
go
deallocate cursor biz_book
go
```

Depending on the content of the row, the user might delete the current row:

```
delete titles where current of biz_book
```

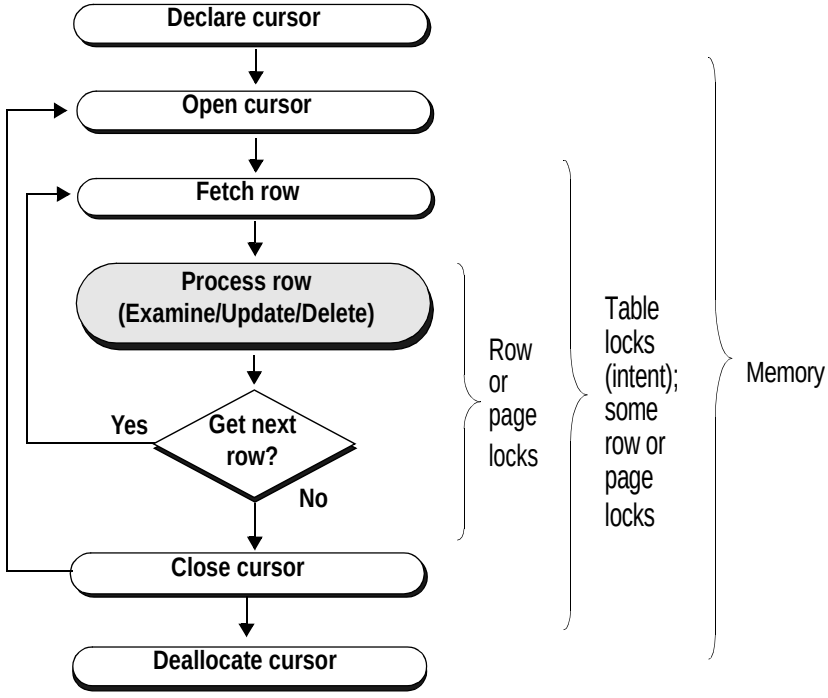
or update the current row:

```
update titles set title="The Rich
    Executive's Database Guide"
where current of biz_book
```

## Resources required at each stage

Cursors use memory and require locks on tables, data pages, and index pages. When you open a cursor, memory is allocated to the cursor and to store the query plan that is generated. While the cursor is open, Adaptive Server holds intent table locks and sometimes row or page locks. Figure 27-3 shows the duration of locks during cursor operations.

***Figure 27-3: Resource use by cursor statement***




The memory resource descriptions in Figure 27-3 and Table 27-1 refer to ad hoc cursors for queries sent by isql or Client-Library™. For other kinds of cursors, the locks are the same, but the memory allocation and deallocation differ somewhat depending on the type of cursor being used, as described in “Memory use and execute cursors” on page 636.

***Table 27-1: Locks and memory use for isql and Client-Library client cursors***

| Cursor command | Resource use |
|---|---|
| declare cursor | When you declare a cursor, Adaptive Server uses only enough memory to store the query text. |
| open | When you open a cursor, Adaptive Server allocates memory to the cursor and to store the query plan that is generated. The server optimizes the query, traverses indexes, and sets up memory variables. The server does not access rows yet, unless it needs to build worktables. However, it does set up the required table-level locks (intent locks). Row and page locking behavior depends on the isolation level, server configuration, and query type.<br>See *System Administration Guide* for more information. |
| fetch | When you execute a fetch, Adaptive Server gets the row(s) required and reads specified values into the cursor variables or sends the row to the client. If the cursor needs to hold lock on rows or pages, the locks are held until a fetch moves the cursor off the row or page or until the cursor is closed. The lock is either a shared or an update lock, depending on how the cursor is written. |
| close | When you close a cursor, Adaptive Server releases the locks and some of the memory allocation. You can open the cursor again, if necessary. |
| deallocate cursor | When you deallocate a cursor, Adaptive Server releases the rest of the memory resources used by the cursor. To reuse the cursor, you must declare it again. |

## Memory use and execute cursors

The descriptions of declare cursor and deallocate cursor in Table 27-1 refer to ad hoc cursors that are sent by isql or Client-Library. Other kinds of cursors allocate memory differently:

- For cursors that are declared *on* stored procedures, only a small amount of memory is allocated at declare cursor time. Cursors declared on stored procedures are sent using Client-Library or the precompiler and are known as execute cursors.
- For cursors declared *within* a stored procedure, memory is already available for the stored procedure, and the declare statement does not require additional memory.

# Cursor modes

There are two cursor modes: read-only and update. As the names suggest, read-only cursors can only display data from a select statement; update cursors can be used to perform positioned updates and deletes.

Read-only mode uses shared page or row locks. If read committed with lock is set to 0, and the query runs at isolation level 1, it uses instant duration locks, and does not hold the page or row locks until the next fetch.

Read-only mode is in effect when you specify for read only or when the cursor's select statement uses distinct, group by, union, or aggregate functions, and in some cases, an order by clause.

Update mode uses update page or row locks. It is in effect when:

- You specify for update.
- The select statement does not include distinct, group by, union, a subquery, aggregate functions, or the at isolation read uncommitted clause.
- You specify shared.

If *column_name_list* is specified, only those columns are updatable.

For more information on locking during cursor processing, see *System Administration Guide*.

Specify the cursor mode when you declare the cursor. If the select statement includes certain options, the cursor is not updatable even if you declare it for update.

# Index use and requirements for cursors

When a query is used in a cursor, it may require or choose different indexes than the same query used outside of a cursor.

## Allpages-locked tables

For read-only cursors, queries at isolation level 0 (dirty reads) require a unique index. Read-only cursors at isolation level 1 or 3 should produce the same query plan as the select statement outside of a cursor.

The index requirements for updatable cursors mean that updatable cursors may use different query plans than read-only cursors. Update cursors have these indexing requirements:

- If the cursor is not declared for update, a unique index is preferred over a table scan or a nonunique index.
- If the cursor is declared for update *without* a for update of list, a unique index is required on allpages-locked tables. An error is raised if no unique index exists.
- If the cursor is declared for update with a for update of list, then only a unique index *without* any columns from the list can be chosen on an allpages-locked table. An error is raised if no unique index qualifies.

When cursors are involved, an index that contains an IDENTITY column is considered unique, even if the index is not declared unique. In some cases, IDENTITY columns must be added to indexes to make them unique, or the optimizer might be forced to choose a suboptimal query plan for a cursor query.

## Data-only-locked tables

In data-only-locked tables, fixed row IDs are used to position cursor scans, so unique indexes are not required for dirty reads or updatable cursors. The only cause for different query plans in updatable cursors is that table scans are used if columns from only useful indexes are included in the for update of list.

### Table scans to avoid the Halloween problem

The Halloween problem is an update anomaly that can occur when a client using a cursor updates a column of the cursor result-set row, and that column defines the order in which the rows are returned from the table. For example, if a cursor was to use an index on last_name, first_name, and update one of these columns, the row could appear in the result set a second time.

To avoid the Halloween problem on data-only-locked tables, Adaptive Server chooses a table scan when the columns from an otherwise useful index are included in the column list of a for update clause.

For implicitly updatable cursors declared without a for update clause, and for cursors where the column list in the for update clause is empty, cursors that update a column in the index used by the cursor may encounter the Halloween problem.

# Comparing performance with and without cursors

This section examines the performance of a stored procedure written two different ways:

- Without a cursor – this procedure scans the table three times, changing the price of each book.
- With a cursor – this procedure makes only one pass through the table.

In both examples, there is a unique index on titles(title_id).

## Sample stored procedure without a cursor

This is an example of a stored procedure without cursors:

```
/* Increase the prices of books in the
** titles table as follows:
**
** If current price is <= $30, increase it by 20%
** If current price is > $30 and <= $60, increase
** it by 10%
** If current price is > $60, increase it by 5%
**
** All price changes must take effect, so this is
** done in a single transaction.
*/

create procedure increase_price
as

    /* start the transaction */
    begin transaction
    /* first update prices > $60 */
    update titles
        set price = price * 1.05
        where price > $60
```

```
    /* next, prices between $30 and $60 */
    update titles
        set price = price * 1.10
    where price > $30 and price <= $60

    /* and finally prices <= $30 */
    update titles
    set price = price * 1.20
    where price <= $30

    /* commit the transaction */
    commit transaction

return
```

## Sample stored procedure with a cursor

This procedure performs the same changes to the underlying table as the procedure written without a cursor, but it uses cursors instead of set-oriented programming. As each row is fetched, examined, and updated, a lock is held on the appropriate data page. Also, as the comments indicate, each update commits as it is made, since there is no explicit transaction.

```
/* Same as previous example, this time using a
** cursor. Each update commits as it is made.
*/
create procedure increase_price_cursor
as
declare @price money

/* declare a cursor for the select from titles */
declare curs cursor for
    select price
    from titles
    for update of price

/* open the cursor */
open curs

/* fetch the first row */
fetch curs into @price

/* now loop, processing all the rows
** @@sqlstatus = 0 means successful fetch
```

```
** @@sqlstatus = 1 means error on previous fetch
** @@sqlstatus = 2 means end of result set reached
*/
while (@@sqlstatus != 2)
begin
    /* check for errors */
    if (@@sqlstatus = 1)
    begin
        print "Error in increase_price"
        return
    end

    /* next adjust the price according to the
    ** criteria
    */
    if @price > $60
    select @price = @price * 1.05
    else
    if @price > $30 and @price <= $60
    select @price = @price * 1.10
    else
    if @price <= $30
    select @price = @price * 1.20

    /* now, update the row */
    update titles
    set price = @price
    where current of curs

    /* fetch the next row */
    fetch curs into @price
end

/* close the cursor and return */
close curs
return
```

Which procedure do you think will have better performance, one that performs three table scans or one that performs a single scan via a cursor?

## Cursor versus noncursor performance comparison

Table 27-2 shows statistics gathered against a 5000-row table. The cursor code takes over 4 times longer, even though it scans the table only once.

***Table 27-2: Sample execution times against a 5000-row table***

| Procedure | Access method | Time |
|---|---|---|
| increase_price | Uses three table scans | 28 seconds |
| increase_price_cursor | Uses cursor, single table scan | 125 seconds |

Results from tests like these can vary widely. They are most pronounced on systems that have busy networks, a large number of active database users, and multiple users accessing the same table.

In addition to locking, cursors involve more network activity than set operations and incur the overhead of processing instructions. The application program needs to communicate with Adaptive Server regarding every result row of the query. This is why the cursor code took much longer to complete than the code that scanned the table three times.

Cursor performance issues include:

- Locking at the page and table level
- Network resources
- Overhead of processing instructions

If there is a set-level programming equivalent, it may be preferable, even if it involves multiple table scans.

# Locking with read-only cursors

Here is a piece of cursor code you can use to display the locks that are set up at each point in the life of a cursor. The following example uses an allpages-locked table. Execute the code in Figure 27-4, and pause at the arrows to execute sp_lock and examine the locks that are in place.

***Figure 27-4: Read-only cursors and locking experiment input***

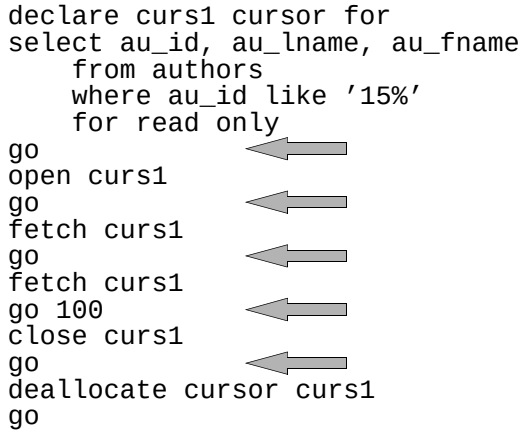



Table 27-3 shows the results.

***Table 27-3: Locks held on data and index pages by cursors***

| Event | Data page |
|---|---|
| After declare | No cursor-related locks. |
| After open | Shared intent lock on authors. |
| After first fetch | Shared intent lock on authors and shared page lock on a page in authors. |
| After 100 fetches | Shared intent lock on authors and shared page lock on a different page in authors. |
| After close | No cursor-related locks. |

If you issue another fetch command after the last row of the result set has been fetched, the locks on the last page are released, so there will be no cursor-related locks.

With a data-only-locked table:

- If the cursor query runs at isolation level 1, and read committed with lock is set to 0, you do not see any page or row locks. The values are copied from the page or row, and the lock is immediately released.
- If read committed with lock is set to 1 or if the query runs at isolation level 2 or 3, you see either shared page or shared row locks at the point that Table 27-3 indicates shared page locks. If the table uses datarows locking, the sp_lock report includes the row ID of the fetched row.

# Isolation levels and cursors

The query plan for a cursor is compiled and optimized when the cursor is opened. You cannot open a cursor and then use set transaction isolation level to change the isolation level at which the cursor operates.

Since cursors using isolation level 0 are compiled differently from those using other isolation levels, you cannot open a cursor at isolation level 0 and open or fetch from it at level 1 or 3. Similarly, you cannot open a cursor at level 1 or 3 and then fetch from it at level 0. Attempts to fetch from a cursor at an incompatible level result in an error message.

Once the cursor has been opened at a particular isolation level, you must deallocate the cursor before changing isolation levels. The effects of changing isolation levels while the cursor is open are as follows:

- Attempting to close and reopen the cursor at another isolation level fails with an error message.
- Attempting to change isolation levels without closing and reopening the cursor has no effect on the isolation level in use and does not produce an error message.

You can include an at isolation clause in the cursor to specify an isolation level. The cursor in the example below can be declared at level 1 and fetched from level 0 because the query plan is compatible with the isolation level:

```
declare cprice cursor for
select title_id, price
    from titles
    where type = "business"
    at isolation read uncommitted
```

# Partitioned heap tables and cursors

A cursor scan of an unpartitioned heap table can read all data up to and including the final insertion made to that table, even if insertions took place after the cursor scan started.

If a heap table is partitioned, data can be inserted into one of the many page chains. The physical insertion point may be before or after the current position of a cursor scan. This means that a cursor scan against a partitioned table is *not* guaranteed to scan the final insertions made to that table.

---

**Note** If your cursor operations require all inserts to be made at the end of a single page chain, *do not* partition the table used in the cursor scan.

---

# Optimizing tips for cursors

Here are several optimizing tips for cursors:

- Optimize cursor selects using the cursor, not an ad hoc query.
- Use union or union all instead of or clauses or in lists.
- Declare the cursor's intent.
- Specify column names in the for update clause.
- Fetch more than one row if you are returning rows to the client.
- Keep cursors open across commits and rollbacks.
- Open multiple cursors on a single connection.

## Optimizing for cursor selects using a cursor

A standalone select statement may be optimized very differently than the same select statement in an implicitly or explicitly updatable cursor. When you are developing applications that use cursors, always check your query plans and I/O statistics using the cursor, rather than using a standalone select. In particular, index restrictions of updatable cursors require very different access methods.

## Using *union* instead of *or* clauses or *in* lists

Cursors cannot use the dynamic index of row IDs generated by the OR strategy. Queries that use the OR strategy in standalone select statements usually perform table scans using read-only cursors. Updatable cursors may need to use a unique index and still require access to each data row, in sequence, in order to evaluate the query clauses.

See "Access Methods and Costing for or and in Clauses" on page 451 for more information.

A read-only cursor using union creates a worktable when the cursor is declared, and sorts it to remove duplicates. Fetches are performed on the worktable. A cursor using union all can return duplicates and does not require a worktable.

## Declaring the cursor's intent

Always declare a cursor's intent: read-only or updatable. This gives you greater control over concurrency implications. If you do not specify the intent, Adaptive Server decides for you, and very often it chooses updatable cursors. Updatable cursors use update locks, thereby preventing other update locks or exclusive locks. If the update changes an indexed column, the optimizer may need to choose a table scan for the query, resulting in potentially difficult concurrency problems. Be sure to examine the query plans for queries that use updatable cursors.

## Specifying column names in the *for update* clause

Adaptive Server acquires update locks on the pages or rows of all tables that have columns listed in the for update clause of the cursor select statement. If the for update clause is not included in the cursor declaration, all tables referenced in the from clause acquire update locks.

The following query includes the name of the column in the for update clause, but acquires update locks only on the titles table, since price is mentioned in the for update clause. The table uses allpages locking. The locks on authors and titleauthor are shared page locks:

```
declare curs3 cursor
for
select au_lname, au_fname, price
    from titles t, authors a,
```

```
        titleauthor ta
where advance <= $1000
    and t.title_id = ta.title_id
    and a.au_id = ta.au_id
for update of price
```

Table 27-4 shows the effects of:

- Omitting the for update clause entirely—no shared clause
- Omitting the column name from the for update clause
- Including the name of the column to be updated in the for update clause
- Adding shared after the name of the titles table while using for update of price

In this table, the additional locks, or more restrictive locks for the two versions of the for update clause are emphasized.

***Table 27-4: Effects of for update clause and shared on cursor locking***

| Clause | *titles* | *authors* | *titleauthor* |
|---|---|---|---|
| None | | sh_page on index | |
| | sh_page on data | sh_page on data | sh_page on data |
| for update | *updpage on index* | *updpage on index* | |
| | updpage on data | *updpage on data* | *updpage on data* |
| for update of price | | sh_page on index | |
| | updpage on data | sh_page on data | sh_page on data |
| for update of price + shared | | sh_page on index | |
| | sh_page on data | sh_page on data | sh_page on data |

## Using *set cursor rows*

The SQL standard specifies a one-row fetch for cursors, which wastes network bandwidth. Using the set cursor rows query option and Open Client's transparent buffering of fetches, you can improve performance:

```
ct_cursor(CT_CURSOR_ROWS)
```

Be careful when you choose the number of rows returned for frequently executed applications using cursors—tune them to the network.

See “Changing network packet sizes” on page 16 for an explanation of this process.

## Keeping cursors open across commits and rollbacks

ANSI closes cursors at the conclusion of each transaction. Transact- SQL provides the set option close on endtran for applications that must meet ANSI behavior. By default, however, this option is turned off. Unless you must meet ANSI requirements, leave this option off to maintain concurrency and throughput.

If you must be ANSI-compliant, decide how to handle the effects on Adaptive Server. Should you perform a lot of updates or deletes in a single transaction? Or should you keep the transactions short?

If you choose to keep transactions short, closing and opening the cursor can affect throughput, since Adaptive Server needs to rematerialize the result set each time the cursor is opened. Choosing to perform more work in each transaction, this can cause concurrency problems, since the query holds locks.

## Opening multiple cursors on a single connection

Some developers simulate cursors by using two or more connections from DB-Library™. One connection performs a select and the other performs updates or deletes on the same tables. This has very high potential to create application deadlocks. For example:

- Connection A holds a shared lock on a page. As long as there are rows pending from Adaptive Server, a shared lock is kept on the current page.
- Connection B requests an exclusive lock on the same pages and then waits.
- The application waits for Connection B to succeed before invoking whatever logic is needed to remove the shared lock. But this never happens.

Since Connection A never requests a lock that is held by Connection B, this is not a server-side deadlock.

CHAPTER 28

# Introduction to Abstract Plans

This chapter provides an overview of abstract plans.



## Definition

Adaptive Server can generate an abstract plan for a query, and save the text and its associated abstract plan in the sysqueryplans system table. Using a rapid hashing method, incoming SQL queries can be compared to saved query text, and if a match is found, the corresponding saved abstract plan is used to execute the query.

An abstract plan describes the execution plan for a query using a language created for that purpose. This language contains operators to specify the choices and actions that can be generated by the optimizer. For example, to specify an index scan on the titles table, using the index title_id_ix, the abstract plan says:

```
( i_scan title_id_ix titles)
```

Abstract plans provide a means for System Administrators and performance tuners to protect the overall performance of a server from changes to query plans. Changes in query plans can arise due to:

- Adaptive Server software upgrades that affect optimizer choices and query plans
- New Adaptive Server features that change query plans

- Changing tuning options such as the parallel degree, table partitioning, or indexing

The major purpose of abstract plans is to provide a means to capture query plans before and after major system changes. The sets of before-and-after query plans can be compared to determine the effects of changes on your queries. Other uses include:

- Searching for specific types of plans, such as table scans or reformatting
- Searching for plans that use particular indexes
- Specifying full or partial plans for poorly-performing queries
- Saving plans for queries with long optimization times

Abstract plans provide an alternative to options that must be specified in the batch or query in order to influence optimizer decisions. Using abstract plans, you can influence the optimization of a SQL statement without having to modify the statement syntax. While matching query text to stored text requires some processing overhead, using a saved plan reduces query optimization overhead.

# Managing abstract plans

A full set of system procedures allows System Administrators and Database Owners to administer plans and plan groups. Individual users can view, drop, and copy the plans for the queries that they have run.

See Chapter 31, “Managing Abstract Plans with System Procedures.”

# Relationship between query text and query plans

For most SQL queries, there are many possible query execution plans. SQL describes the desired result set, but does not describe how that result set should be obtained from the database. Consider a query that joins three tables, such as this:

```
select t1.c11, t2.c21
from t1, t2, t3
```

```
where t1.c11 = t2.c21
and t1.c11 = t3.c31
```

There are many different possible join orders, and depending on the indexes that exist on the tables, many possible access methods, including table scans, index scans, and the reformatting strategy. Each join may use either a nested-loop join or a merge join. These choices are determined by the optimizer's query costing algorithms, and are not included in or specified in the query itself.

When you capture the abstract plan, the query is optimized in the usual way, except that the optimizer also generates an abstract plan, and saves the query text and abstract plan in sysqueryplans.

## Limits of options for influencing query plans

Adaptive Server provides other options for influencing optimizer choices:

- Session-level options such as set forceplan to force join order or set parallel_degree to specify the maximum number of worker processes to use for the query
- Options that can be included in the query text to influence the index choice, cache strategy, and parallel degree

There are some limitations to using set commands or adding hints to the query text:

- Not all query plan steps can be influenced, for example, subquery attachment
- Some query-generating tools do not support the in-query options or require all queries to be vendor-independent

# Full versus partial plans

Abstract plans can be full plans, describing all query processing steps and options, or they can be partial plans. A partial plan might specify that an index is to be used for the scan of a particular table, without specifying the index name or the join order for the query. For example:

```
select t1.c11, t2.c21
from t1, t2, t3
```

```
where t1.c11 = t2.c21
and t1.c11 = t3.c31
```

The full abstract plan includes:

- The join type, either nl_g_join for nested-loop joins, or m_g_join for merge joins. The plan for this query specifies a nested-loop join.
- The join order, included in the nl_g_join clause.
- The type of scan, t_scan for table scan or i_scan for index scan.
- The name of the index chosen for the tables that are accessed via an index scan.
- The scan properties: the parallel degree, I/O size, and cache strategy for each table in the query.

The abstract plan for the query above specifies the join order, the access method for each table in the query, and the scan properties for each table:

```
( nl_g_join
    ( t_scan t2 )
    ( i_scan t1_c11_ix t1 )
    ( i_scan t3_c31_ix t3 )
)
( prop t3
    ( parallel 1 )
    ( prefetch 16 )
    ( lru )
)
( prop t1
    ( parallel 1 )
    ( prefetch 16 )
    ( lru )
)
( prop t2
    ( parallel 1 )
    ( prefetch 16 )
    ( lru )
)
```

Chapter 32, “Abstract Plan Language Reference,” provides a reference to the abstract plan language and syntax.

## Creating a partial plan

When abstract plans are captured, full abstract plans are generated and stored. You can write partial plans to affect only a subset of the optimizer choices. If the query above had not used the index on t3, but all other parts of the query plan were optimal, you could create a partial plan for the query using the create plan command. This partial plan specifies only the index choice for t3:

```
create plan
"select t1.c11, t2.c21
from t1, t2, t3
where t1.c11 = t2.c21
and t1.c11 = t3.c31"
"( i_scan t3_c31_ix t3 )"
```

You can also create abstract plans with the plan clause for select, delete, update, and other commands that can be optimized.

See “Creating plans using SQL” on page 690.

# Abstract plan groups

When you first install Adaptive Server, there are two abstract plan groups:

- ap_stdout, used by default for capturing plans
- ap_stdin, used by default for plan association

A System Administrator can enable server-wide plan capture to ap_stdout, so that all query plans for all queries are captured. Server-wide plan association uses queries and plans from ap_stdin. If some queries require specially-tuned plans, they can be made available server-wide.

A System Administrator or Database Owner can create additional plan groups, copy plans from one group to another, and compare plans in two different groups.

The capture of abstract plans and the association of abstract plans with queries always happens within the context of the currently-active plan group. Users can use session-level set commands to enable plan capture and association.

Some of the ways abstract plan groups can be used are:

- A query tuner can create abstract plans in a group created for testing purposes without affecting plans for other users on the system
- Using plan groups, “before” and “after” sets of plans can be used to determine the effects of system or upgrade changes on query optimization.

See Chapter 30, “Creating and Using Abstract Plans,” for information on enabling the capture and association of plans.

# How abstract plans are associated with queries

When an abstract plan is saved, all white space (returns, tabs, and multiple spaces) in the query is trimmed to a single space, and a hash-key value is computed for the white-space trimmed SQL statement. The trimmed SQL statement and the hash key are stored in sysqueryplans along with the abstract plan, a unique plan ID, the user’s ID, and the ID of the current abstract plan group.

When abstract plan association is enabled, the hash key for incoming SQL statements is computed, and this value is used to search for the matching query and abstract plan in the current association group, with the corresponding user ID. The full **association key** of an abstract plans consists of:

- The user ID of the current user
- The group ID of the current association group
- The full query text

Once a matching hash key is found, the full text of the saved query is compared to the query to be executed, and used if it matches.

The association key combination of user ID, group ID and query text means that for a given user, there cannot be two queries in the same abstract plan group that have the same query text, but different query plans.

CHAPTER 29

# Abstract Query Plan Guide

This chapter covers some guidelines you can use in writing Abstract Plans.



## Introduction

Abstract plans allow you to specify the desired execution plan of a query. Abstract plans provide an alternative to the session-level and query level options that force a join order, or specify the index, I/O size, or other query execution options. The session-level and query-level options are described in Chapter 30, “Creating and Using Abstract Plans.”

There are several optimization decisions that cannot be specified with set commands or clauses included in the query text. Some examples are:

- Subquery attachment
- The join order for flattened subqueries
- Reformatting

In many cases, including set commands or changing the query text is not always possible or desired. Abstract plans provide an alternative, more complete method of influencing optimizer decisions.

Abstract plans are relational algebra expressions that are not included in the query text. They are stored in a system catalog and associated to incoming queries based on the text of these queries.

The tables used in this section are the same as those in Chapter 32, “Abstract Plan Language Reference.” See “Schema for examples” on page 712 for the create table and create index statements.

## Abstract plan language

The abstract plan language is a relational algebra that uses these operators:

- g_join, the generic join, a high-level logical join operator. It describes inner, outer and existence joins, using either nested-loop joins or sort-merge joins.
- nl_g_join, specifying a nested-loop join, including all inner, outer, and existence joins
- m_g_join, specifying a merge join, including inner and outer joins.
- union, a logical union operator. It describes both the union and the union all SQL constructs.
- scan, a logical operator that transforms a stored table in a flow of rows, a *derived table*. It allows partial plans that do not restrict the access method.
- i_scan, a physical operator, implementing scan. It directs the optimizer to use an index scan on the specified table.
- t_scan, a physical operator, implementing scan. It directs the optimizer to use a full table scan on the specified table.
- store, a logical operator, describing the materialization of a derived table in a stored worktable.
- nested, a filter, describing the placement and structure of nested subqueries.

See “Schema for examples” on page 712 for the create table and create index commands used for the examples in this section.

Additional abstract plan keywords are used for grouping and identification:

- plan groups the elements when a plan requires multiple steps.
- hints groups a set of hints for a partial plan.
- prop introduces a set of scan properties for a table: prefetch, lru|mru and parallel.

- table identifies a table when correlation names are used, and in subqueries or views.
- work_t identifies a worktable.
- in, used with table, for identifying tables named in a subquery (subq) or view (view).
- subq is also used under the nested operator to indicate the attachment point for a nested subquery, and to introduce the subqueries abstract plan.

## Queries, access methods, and abstract plans

For any specific table, there can be several access methods for a specific query: index scans using different indexes, table scans, the OR strategy, and reformatting are some examples.

This simple query has several choices of access methods:

```
select * from t1
where c11 > 1000 and c12 < 0
```

The following abstract plans specify three different access methods:

- Use the index i_c11:

  ```
  (i_scan i_c11 t1)
  ```

- Use the index i_c12:

  ```
  (i_scan i_c12 t1)
  ```

- Do a full table scan:

  ```
  (t_scan t1)
  ```

Abstract plans can be full plans, specifying all optimizer choices for a query, or can specify a subset of the choices, such as the index to use for a single table in the query, but not the join order for the tables. For example, using a partial abstract plan, you can specify that the query above should use some index and let the optimizer choose between i_c11 and i_c12, but not do a full table scan. The empty parentheses are used in place of the index name:

```
(i_scan () t1)
```

In addition, the query could use either 2K or 16K I/O, or be performed in serial or parallel.

## Identifying tables

Abstract plans need to name all of a query's tables in a non-ambiguous way, such that a table named in the abstract can be linked to its occurrence in the SQL query. In most cases, the table name is all that is needed. If the query qualifies the table name with the database and owner name, these are also needed to fully identify a table in the abstract plan. For example, this example used the unqualified table name:

```
select * from t1
```

The abstract plan also uses the unqualified name:

```
(t_scan t1)
```

If a database name and/or owner name are provided in the query:

```
select * from pubs2.dbo.t1
```

Then the abstract plan must also use the qualifications:

```
(t_scan pubs2.dbo.t1)
```

However, the same table may occur several times in the same query, as in this example:

```
select * from t1 a, t1 b
```

Correlation names, `a` and `b` in the example above, identify the two tables in SQL. In an abstract plan, the `table` operator associates each correlation name with the occurrence of the table:

```
( g_join
        ( t_scan ( table ( a t1 ) ) )
        ( t_scan ( table ( b t1 ) ) )
)
```

Table names can also be ambiguous in views and subqueries, so the `table` operator is used for tables in views and subqueries.

For subqueries, the `in` and `subq` operators qualify the name of the table with its syntactical containment by the subquery. The same table is used in the outer query and the subquery in this example:

```
select *
from t1
where c11 in (select c12 from t1 where c11 > 100)
```

The abstract plan identifies them unambiguously:

```
( g_join
    ( t_scan t1 )
```

```
    ( i_scan i_c11_c12 ( table t1 ( in ( subq 1 ) ) ) )
)
```

For views, the in and view operators provide the identification. The query in this example references a table used in the view:

```
create view v1
as
select * from t1 where c12 > 100
select t1.c11 from t1, v1
    where t1.c12 = v1.c11
```

Here is the abstract plan:

```
( g_join
    ( t_scan t1 )
    ( i_scan i_c12 ( table t1 ( in ( view v1 ) ) ) )
)
```

## Identifying indexes

The i_scan operator requires two operands, the index name and the table name, as shown here:

```
( i_scan i_c12 t1 )
```

To specify that some index should be used, without specifying the index, substitute empty parenthesis for the index name:

```
( i_scan ( ) t1 )
```

## Specifying join order

Adaptive Server performs joins of three or more tables by joining two of the tables, and joining the “derived table” from that join to the next table in the join order. This derived table is a flow of rows, as from an earlier nested-loop join in the query execution.

This query joins three tables:

```
select *
from t1, t2, t3
where c11 = c21
    and c12 = c31
    and c22 = 0
    and c32 = 100
```

This example shows the binary nature of the join algorithm, using g_join operators. The plan specifies the join order t2, t1, t3:

```
(g_join
    (g_join
        (scan t2)
        (scan t1)
    )
    (scan t3)
)
```

The results of the t2-t1 join are then joined to t3. The scan operator in this example leaves the choice of table scan or index scan up to the optimizer.

## Shorthand notation for joins

In general, a *N*-way join, with the order t1, t2, t3..., tN-1, *tN* is described by:

```
(g_join
    (g_join
        ...
            (g_join
                (g_join
                    (scan t1)
                    (scan t2)
                )
                (scan t3)
            )
        ...
        (scan tN-1)
    )
    (scan tN)
)
```

This notation can be used as shorthand for the g_join operator:

```
(g_join
    (scan t1)
    (scan t2)
    (scan t3)
    ...
    (scan tN-1)
    (scan tN)
)
```

This notation can be used for g_join, and nl_g_join, and m_g_join.

## Join order examples

The optimizer could select among several plans for this three-way join query:

```
select *
from t1, t2, t3
where c11 = c21
    and c12 = c31
    and c22 = 0
    and c32 = 100
```

Here are a few examples:

- Use c22 as a search argument on t2, join with t1 on c11, then with t3 on c31:

```
(g_join
    (i_scan i_c22 t2)
    (i_scan i_c11 t1)
    (i_scan i_c31 t3)
)
```

- Use the search argument on t3, and the join order t3, t1, t2:

```
(g_join
    (i_scan i_c32 t3)
    (i_scan i_c12 t1)
    (i_scan i_c21 t2)
)
```

- Do a full table scan of t2, if it is small and fits in cache, still using the join order t3, t1, t2:

```
(g_join
    (i_scan i_c32 t3)
    (i_scan i_c12 t1)
    (t_scan t2)
)
```

- If t1 is very large, and t2 and t3 individually qualify a large part of t1, but together a very small part, this plan specifies a STAR join:

```
(g_join
    (i_scan i_c22 t2)
    (i_scan i_c32 t3)
    (i_scan i_c11_c12 t1)
)
```

All of these plans completely constrain the choice of join order, letting the optimizer choose the type of join.

The generic g_join operator implements outer joins, inner joins, and existence joins. For examples of flattened subqueries that perform existence joins, see “Flattened subqueries” on page 668.

## Match between execution methods and abstract plans

There are some limits to join orders and join types, depending on the type of query. One example is outer joins, such as:

```
select * from t1, t2
where c11 *= c21
```

Adaptive Server requires the outer member of the outer join to be the outer table during join processing. Therefore, this abstract plan is illegal:

```
(g_join
    (scan t2)
    (scan t1)
)
```

Attempting to use this plan results in an error message, and the query is not compiled.

## Specifying join order for queries using views

You can use abstract plans to enforce the join order for merged views. This example creates a view. This view performs a join of t2 and t3:

```
create view v2
as
select *
from t2, t3
where c22 = c32
```

This query performs a join with the t2 in the view:

```
select * from t1, v2
where c11 = c21
    and c22 = 0
```

This abstract plan specifies the join order t2, t1, t3:

```
(g_join
    (scan (table t2 (in (view v2))))
    (scan t1)
    (scan (table t3 (in (view v2))))
)
```

This example joins with *t3* in the view:

```
select * from t1, v2
where c11 = c31
    and c32 = 100
```

This plan uses the join order t3, t1, t2:

```
(g_join
    (scan (table t3 (in (view v2))))
    (scan t1)
    (scan (table t2 (in (view v2))))
)
```

This is an example where abstract plans can be used, if needed, to affect the join order for a query, when set forceplan cannot.

## Specifying the join type

Adaptive Server can perform either nested-loop or merge joins. The g_join operator leaves the optimizer free to choose the best join algorithm, based on costing. To specify a nested-loop join, use the nl_g_join operator; for a sort-merge join, use the m_g_join operator. Abstract plans captured by Adaptive Server always include the operator that specifies the algorithm, and not the g_join operator.

Note that the "g" that appears in each operator means "generic," meaning that they apply to inner joins and outer joins; g_join and nl_g_join can also apply to existence joins.

This query specifies a join between t1 and t2:

```
select * from t1, t2
    where c12 = c21 and c11 = 0
```

This abstract plan specifies a nested-loop join:

```
(nl_g_join
    (i_scan i_c11 t1)
    (i_scan i_c21 t2)
)
```

The nested-loop plan uses the index i_c11 to limit the scan using the search clause, and then performs the join with t2, using the index on the join column.

This merge-join plan uses different indexes:

```
(m_g_join
    (i_scan i_c12 t1)
```

```
        (i_scan i_c21 t2)
    )
```

The merge join uses the indexes on the join columns, i_c12 and i_c21, for the merge keys. This query performs a full-merge join and no sort is needed.

A merge join could also use the index on i_c11 to select the rows from t1 into a worktable; the merge uses the index on i_c21:

```
(m_g_join
    (i_scan i11 t1)
    (i_scan i21 t2)
)
```

The step that creates the worktable is not specified in the plan; the optimizer detects when a worktable and sort are needed for join-key ordering.

## Specifying partial plans and hints

There are cases when a full plan is not needed. For example, if the only problem with a query plan is that the optimizer chooses a table scan instead of using a nonclustered index, the abstract plan can specify only the index choice, and leave the other decisions to the optimizer.

The optimizer could choose a table scan of t3 rather than using i_c31 for this query:

```
select *
from t1, t2, t3
where c11 = c21
    and c12 < c31
    and c22 = 0
    and c32 = 100
```

The following plan, as generated by the optimizer, specifies join order t2, t1, t3. However, the plan specifies a table scan of t3:

```
(g_join
    (i_scan i_c22 t2)
    (i_scan i_c11 t1)
    (t_scan t3)
)
```

This full plan could be modified to specify the use of i_c31 instead:

```
(g_join
```

```
    (i_scan i_c22 t2)
    (i_scan i_c11 t1)
    (i_scan i_c31 t3)
)
```

However, specifying only a partial abstract plan is a more flexible solution. As data in the other tables of that query evolves, the optimal join order can change. The partial plan can specify just one partial plan item. For the index scan of t3, the partial plan is simply:

```
(i_scan i_c31 t3)
```

The optimizer chooses the join order and the access methods for t1 and t2.

## Grouping multiple hints

There may be cases where more than one plan fragment is needed. For example, you might want to specify that some index should be used for each table in the query, but leave the join order up to the optimizer. When multiple hints are needed, they can be grouped with the hints operator:

```
(hints
    (i_scan () t1)
    (i_scan () t2)
    (i_scan () t3)
)
```

In this case, the role of the hints operator is purely syntactic; it does not affect the ordering of the scans.

There are no limits on what may be given as a hint. Partial join orders may be mixed with partial access methods. This hint specifies that t2 is outer to t1 in the join order, and that the scan of t3 should use an index, but the optimizer can choose the index for t3, the access methods for t1 and t2, and the placement of t3 in the join order:

```
(hints
    (g_join
        (scan t2)
        (scan t1)
    )
    (i_scan () t3)
)
```

### Inconsistent and illegal plans using hints

It is possible to describe inconsistent plans using hints, such as this plan that specifies contradictory join orders:

```
(hints
    (g_join
        (scan t2)
        (scan t1)
    )
    (g_join
        (scan t1)
        (scan t2)
    )
)
```

When the query associated with the plan is executed, the query cannot be compiled, and an error is raised.

Other inconsistent hints do not raise an exception, but may use any of the specified access methods. This plan specifies both an index scan and a table scan for the same table:

```
(hints
    (t_scan t3)
    (i_scan () t3)
)
```

In this case, either method may be chosen, the behavior is indeterminate.

## Creating abstract plans for subqueries

Subqueries are resolved in several ways in Adaptive Server, and the abstract plans reflect the query execution steps:

- Materialization – The subquery is executed and results are stored in a worktable or internal variable. See “Materialized subqueries” on page 667.
- Flattening – The query is flattened into a join with the tables in the main query. See “Flattened subqueries” on page 668.
- Nesting – The subquery is executed once for each outer query row. See “Nested subqueries” on page 669.

Abstract plans do not allow the choice of the basic subquery resolution method. This is a rule-based decision and cannot be changed during query optimization. Abstract plans, however, can be used to influence the plans for the outer and inner queries. In nested subqueries, abstract plans can also be used to choose where the subquery is nested in the outer query.

## Materialized subqueries

This query includes a non correlated subquery that can be materialized:

```
select *
from t1
where c11 = (select count(*) from t2)
```

The first step in the abstract plan materializes the scalar aggregate in the subquery. The second step uses the result to scan t1:

```
( plan
    ( i_scan i_c21 ( table t2 ( in (subq 1 ) ) ) )
    ( i_scan i_c11 t1 )
)
```

This query includes a vector aggregate in the subquery:

```
select *
from t1
where c11 in (select max(c21)
        from t2
        group by c22)
```

The abstract plan materializes the subquery in the first step, and joins it to the outer query in the second step:

```
( plan
    ( store Worktab1
        ( t_scan ( table t2 ( in (subq 1 ) ) ) )
    )
    ( nl_g_join
        ( t_scan t1 )
        ( t_scan ( work_t Worktab1 ) )
    )
)
```

## Flattened subqueries

Some subqueries can be flattened into joins. The g_join and nl_g_join operators leave it to the optimizer to detect when an existence join is needed. For example, this query includes a subquery introduced with exists:

```
select * from t1
where c12 > 0
    and exists (select * from t2
            where t1.c11 = c21
                and c22 < 100)
```

The semantics of the query require an existence join between t1 and t2. The join order t1, t2 is interpreted by the optimizer as an existence join, with the scan of t2 stopping on the first matching row of t2 for each qualifying row in t1:

```
(g_join
    (scan t1)
    (scan (table t2 (in (subq 1) ) ))
)
```

The join order t2, t1 requires other means to guarantee the duplicate elimination:

```
(g_join
    (scan (table t2 (in (subq 1) ) ) )
    (scan t1)
)
```

Using this abstract plan, the optimizer can decide to use:

- A unique index on t2.c21, if one exists, with a regular join.
- The unique reformatting strategy, if no unique index exists. In this case, the query will probably use the index on c22 to select the rows into a worktable.
- The duplicate elimination sort optimization strategy, performing a regular join and selecting the results into the worktable, then sorting the worktable.

The abstract plan does not need to specify the creation and scanning of the worktables needed for the last two options.

For more information on subquery flattening, see “Flattening in, any, and exists subqueries” on page 494.

### Example: changing the join order in a flattened subquery

The query can be flattened to an existence join:

```
select *
from t1, t2
where c11 = c21
    and c21 > 100
    and exists (select * from t3
        where c31 != t1.c11)
```

The "!=" correlation can make the scan of t3 rather expensive. If the join order is t1, t2, the best place for t3 in the join order depends on whether the join of t1 and t2 increases or decreases the number of rows, and therefore, the number of times that the expensive table scan needs to be performed. If the optimizer fails to find the right join order for t3, the following abstract plan can be used when the join reduces the number of times that t3 must be scanned:

```
(g_join
    (scan t1)
    (scan t2)
    (scan (table t3 (in (subq 1) ) ) )
)
```

If the join increases the number of times that t3 needs to be scanned, this abstract plan performs the scans of t3 before the join:

```
(g_join
    (scan t1)
    (scan (table t3 (in (subq 1) ) ) )
    (scan t2)
)
```

### Nested subqueries

Nested subqueries can be explicitly described in abstract plans:

- The abstract plan for the subquery is provided.
- The location at which the subquery attaches to the main query is specified.

Abstract plans allow you to affect the query plan for the subquery, and to change the attachment point for the subquery in the outer query.

The nested operator specifies the position of the subquery in the outer query. Subqueries are "nested over" a specific derived table. The optimizer chooses a spot where all the correlation columns for the outer query are available, and where it estimates that the subquery needs to be executed the least number of times.

The following SQL statement contains a correlated expression subquery:

```
select *
from t1, t2
where c11 = c21
    and c21 > 100
    and c12 = (select c31 from t3
                where c32 = t1.c11)
```

The abstract plan shows the subquery nested over the scan of t1:

```
( g_join
    ( nested
        ( i_scan i_c12 t1 )
        ( subq 1
            (t_scan ( table t3 ( in ( subq 1 ) ) ) )
         )
    )
    ( i_scan i_c21 t2 )
)
```

## Subquery identification and attachment

Subqueries are identified with numbers, in the order of their leading opened parenthesis "(".

This example has two subqueries, one in the select list:

```
select
    (select c11 from t1 where c12 = t3.c32), c31
from t3
where c32 > (select c22 from t2 where c21 = t3.c31)
```

In the abstract plan, the subquery containing t1 is named "1" and the subquery containing t2 is named "2". Both subquery 1 and 2 are nested over the scan of t3:

```
( nested
    ( nested
        ( t_scan t3 )
        ( subq 1
           ( i_scan i_c11_c12 ( table t1 (in ( subq 1 ) ) ) )
```

```
            )
        )
        ( subq 2
            ( i_scan i_c21 ( table t2 ( in ( subq 2 ) ) ) )
        )
    )
```

In this query, the second subquery is nested in the first:

```
select * from t3
where c32 > all
    (select c11 from t1 where c12 > all
        (select c22 from t2 where c21 = t3.c31))
```

In this case, the subquery containing t1 is also named "1" and the subquery containing t2 is named "2". In this plan, subquery 2 is nested over the scan of t1, which is performed in subquery 1; subquery 1 is nested over the scan of t3 in the main query:

```
( nested
    ( t_scan t3 )
    ( subq 1
        ( nested
            ( i_scan i_c11_c12 ( table t1 ( in ( subq 1 ) ) ) )
            ( subq 2
                ( i_scan i_c21 ( table t2 ( in ( subq 2 ) ) ) )
            )
        )
    )
```

## More subquery examples: reading ordering and attachment

The nested operator has the derived table as the first operand and the nested subquery as the second operand. This allows an easy vertical reading of the join order and subquery placement:

```
select *
from t1, t2, t3
where c12 = 0
    and c11 = c21
    and c22 = c32
    and 0 < (select c21 from t2 where c22 = t1.c11)
```

In the plan, the join order is t1, t2, t3, with the subquery nested over the scan of t1:

```
( g_join
    ( nested
```

```
            ( i_scan i_c11 t1 )
            ( subq 1
               ( t_scan ( table t2 ( in (subq 1 ) ) ) )
            )
        )
        ( i_scan i_c21 t2 )
        ( i_scan i_c32 t3 )
    )
```

## Modifying subquery nesting

If you modify the attachment point for a subquery, you must choose a point at which all of the correlation columns are available.This query is correlated to both of the tables in the outer query:

```
select *
from t1, t2, t3
where c12 = 0
    and c11 = c21
    and c22 = c32
    and 0 < (select c31 from t3 where c31 = t1.c11
                    and c32 = t2.c22)
```

This plan uses the join order t1, t2, t3, with the subquery nested over the t1-t2 join:

```
( g_join
    ( nested
        ( g_join
            ( i_scan i_c11_c12 t1 )
            ( i_scan i_c22 t2 )
        )
        ( subq 1
           ( t_scan ( table t3 ( in (subq 1 ) ) ) )
        )
    )
    ( i_scan i_c32 t3 )
)
```

Since the subquery requires columns from both outer tables, it would be incorrect to nest it over the scan of t1 or the scan of t2; such errors are silently corrected during optimization.

## Abstract plans for materialized views

This view is materialized during query processing:

```
create view v3
as
select distinct *
from t3
```

This query performs a join with the materialized view:

```
select *
from t1, v3
where c11 = c31
```

A first step materializes the view `v3` into a worktable. The second joins it with the main query table `t1` :

```
( plan
        ( store Worktab1
              ( t_scan ( table t3 ( in (view v3 ) ) ) )
        )
        ( g_join
              ( t_scan t1 )
              ( t_scan ( work_t Worktab1 ) )
        )
)
```

## Abstract plans for queries containing aggregates

This query returns a scalar aggregate:

```
select max(c11) from t1
```

The first step computes the scalar aggregate and stores it in an internal variable. The second step is empty, as it only returns the variable, in a step with nothing to optimize:

```
( plan
        ( t_scan t1 )
        ( )
)
```

Vector aggregates are also two-step queries:

```
select max(c11)
from t1
group by c12
```

The first step processes the aggregates into a worktable; the second step scans the worktable:

```
( plan
        ( store Worktab1
                ( t_scan t1 )
        )
        ( t_scan ( work_t Worktab1 ) )
)
```

Nested aggregates are a Transact-SQL extension:

```
select max(count(*))
from t1
group by c11
```

The first step processes the vector aggregate into a worktable, the second scans it to process the nested scalar aggregate into an internal variable, and the third step returns the value.

```
( plan
     ( store Worktab1
          ( i_scan i_c12 t1 )
     )
     ( t_scan ( work_t Worktab1 ) )
     ( )
)
```

Extended columns in aggregate queries are a Transact-SQL extension:

```
select max(c11), c11
from t1
group by c12
```

The first step processes the vector aggregate; the second one joins it back to the base table to process the extended columns:

```
( plan
     ( store Worktab1
          ( t_scan t1 )
     )
     ( g_join
          ( t_scan t1 )
          ( i_scan i_c11 ( work_t Worktab1 ) )
     )
)
```

This example contains an aggregate in a merged view:

```
create view v4
```

```
as
select max(c11) as c41, c12 as c42
from t1
group by c12
select * from t2, v4
where c21 = 0
    and c22 > c41
```

The first step processes the vector aggregate; the second joins it to the main query table:

```
( plan
    ( store Worktab1
        ( t_scan ( table t1 ( in (view v4 ) ) ) )
    )
    ( g_join
        ( i_scan i_c22 t2 )
        ( t_scan ( work_t Worktab1 ) )
    )
)
```

This example includes an aggregate that is processed using a materialized view:

```
create view v5
as
select distinct max(c11) as c51, c12 as c52
from t1
group by c12
select * from t2, v5
where c21 = 0
    and c22 > c51
```

The first step processes the vector aggregate into a worktable. The second step scans it into a second worktable to process the materialized view. The third step joins this second worktable in the main query:

```
( plan
    ( store Worktab1
        ( t_scan ( table t1 ( in (view v5 ) ) ) )
    )
    ( store Worktab2
        ( t_scan ( work_t Worktab1 ) )
    )
    ( g_join
        ( i_scan i_c22 t2 )
        ( t_scan ( work_t Worktab2 ) )
    )
```

```
)
```

## Specifying the reformatting strategy

In this query, t2 is very large, and has no index:

```
select *
from t1, t2
where c11 > 0
    and c12 = c21
    and c22 = 0
```

The abstract plan that specifies the reformatting strategy on t2 is:

```
( g_join
    (t_scan t1
    (scan
        (store Worktab1
            (t_scan t2)
        )
    )
)
```

In the case of the reformatting strategy, the store operator is an operand of scan. This is the only case when the store operator is not the operand of a plan operator.

## OR strategy limitation

The OR strategy has no matching abstract plan that describes the RID scan required to perform the final step. All abstract plans generated by Adaptive Server for the OR strategy specify only the scan operator. You cannot use abstract plans to influence index choice for queries that require the OR strategy to eliminate duplicates.

## When the *store* operator is not specified

Some multistep queries that require worktables do not require multistep plans with a separate worktable step, and the use of the store operator to create the worktable. These are:

- The sort step of queries using distinct

- The worktables needed for merge joins
- Worktables needed for union queries
- The sort step, when a flattened subquery requires sort to remove duplicates

# Tips on writing abstract plans

Here are some additional tips for writing and using abstract plans:

- Look at the current plan for the query and at plans that use the same query execution steps as the plan you need to write. It is often easier to modify an existing plan than to write a full plan from scratch.
  - Capture the plan for the query.
  - Use sp_help_qplan to display the SQL text and plan.
  - Edit this output to generate a create plan command, or attach an edited plan to the SQL query using the plan clause.
- It is often best to specify partial plans for query tuning in cases where most optimizer decisions are appropriate, but only an index choice, for example, needs improvement.

  By using partial plans, the optimizer can choose other paths for other tables as the data in other tables changes.
- Once saved, abstract plans are static. Data volumes and distributions may change so that saved abstract plans are no longer optimal.

  Subsequent tuning changes made by adding indexes, partitioning a table, or adding buffer pools may mean that some saved plans are not performing as well as possible under current conditions. Most of the time, you want to operate with a small number of abstract plans that solve specific problems.

  Perform periodic plan checks to verify that the saved plans are still better than the plan that the optimizer would choose.

# Comparing plans “before” and “after”

Abstract query plans can be used to assess the impact of an Adaptive Server software upgrade or system tuning changes on your query plans. You need to save plans before the changes are made, perform the upgrade or tuning changes, and then save plans again and compare the plans. The basic set of steps is:

1 Enable server-wide capture mode by setting the configuration parameter abstract plan dump to 1. All plans are then captured in the default group, ap_stdout.

2 Allow enough time for the captured plans to represent most of the queries run on the system. You can check whether additional plans are being generated by checking whether the count of rows in the ap_stdout group in sysqueryplans is stable:

```
select count(*) from sysqueryplans where gid = 2
```

3 Copy all plans from ap_stdout to ap_stdin (or some other group, if you do not want to use server-wide plan load mode), using sp_copy_all_qplans.

4 Drop all query plans from ap_stdout, using sp_drop_all_qplans.

5 Perform the upgrade or tuning changes.

6 Allow sufficient time for plans to be captured to ap_stdout.

7 Compare plans in ap_stdout and ap_stdin, using the diff mode parameter of sp_cmp_all_qplans. For example, this query compares all plans in ap_stdout and ap_stdin:

```
sp_cmp_all_qplans ap_stdout, ap_stdin, diff
```

This displays only information about the plans that are different in the two groups.

## Effects of enabling server-wide capture mode

When server-wide capture mode is enabled, plans for all queries that can be optimized are saved in all databases on the server. Some possible system administration impacts are:

- When plans are captured, the plan is saved in sysqueryplans and log records are generated. The amount of space required for the plans and log records depends on the size and complexity of the SQL statements and query plans. Check space in each database where users will be active.

  You may need to perform more frequent transaction log dumps, especially in the early stages of server-wide capture when many new plans are being generated.

- If users execute system procedures from the master database, and installmaster was loaded with server-wide plan capture enabled, then plans for the statements that can be optimized in system procedures are saved in master..sysqueryplans.

  This is also true for any user-defined procedures created while plan capture was enabled. You may want to provide a default database at login for all users, including System Administrators, if space in master is limited.

- The sysqueryplans table uses datarows locking to reduce lock contention. However, especially when a large number of new plans are being saved, there may be a slight impact on performance.

- While server-wide capture mode is enabled, using bcp saves query plans in the master database. If you perform bcp using a large number of tables or views, check sysqueryplans and the transaction log in master.

## Time and space to copy plans

If you have a large number of query plans in ap_stdout, be sure there is sufficient space to copy them on the system segment before starting the copy. Use sp_spaceused to check the size of sysqueryplans, and sp_helpsegment to check the size of the system segment.

Copying plans also requires space in the transaction log.

sp_copy_all_qplans calls sp_copy_qplan for each plan in the group to be copied. If sp_copy_all_qplans fails at any time due to lack of space or other problems, any plans that were successfully copied remain in the target query plan group.

# Abstract plans for stored procedures

For abstract plans to be captured for the SQL statements that can be optimized in stored procedures:

- The procedures must be created while plan capture or plan association mode is enabled. (This saves the text of the procedure in sysprocedures.)
- The procedure must be executed with plan capture mode enabled, and the procedure must be read from disk, not from the procedure cache.

This sequence of steps captures the query text and abstract plans for all statements in the procedure that can be optimized:

```
set plan dump dev_plans on
go
create procedure myproc as ...
go
exec myproc
go
```

If the procedure is in cache, so that the plans for the procedure are not being captured, you can execute the procedure with recompile. Similarly, once a stored procedure has been executed using an abstract query plan, the plan in the procedure cache is used so that query plan association does not take place unless the procedure is read from disk.

## Procedures and plan ownership

When plan capture mode is enabled, abstract plans for the statements in a stored procedure that can be optimized are saved with the user ID of the owner of the procedure.

During plan association mode, association for stored procedures is based on the user ID of the owner of the procedure, not the user who executes the procedure. This means that once an abstract query plan is created for a procedure, all users who have permission to execute the procedure use the same abstract plan.

## Procedures with variable execution paths and optimization

Executing a stored procedure saves abstract plans for each statement that can be optimized, even if the stored procedure contains control-of-flow statements that can cause different statements to be run depending on parameters to the procedure or other conditions. If the query is run a second time with different parameters that use a different code path, plans for any statements that were optimized and saved by the earlier execution, and the abstract plan for the statement is associated with the query.

However, abstract plans for procedures do not solve the problem with procedures with statements that are optimized differently depending on conditions or parameters. One example is a procedure where users provide the low and high values for a between clause, with a query such as:

```
select title_id
from titles
where price between @lo and @hi
```

Depending on the parameters, the best plan could either be index access or a table scan. For these procedures, the abstract plan may specify either access method, depending on the parameters when the procedure was first executed. For more information on optimization of procedures, see “Splitting stored procedures to improve costing” on page 403.

# Ad Hoc queries and abstract plans

Abstract plan capture saves the full text of the SQL statement and abstract plan association is based on the full text of the SQL query. If users submit ad hoc SQL statements, rather than using stored procedures or Embedded SQL, abstract plans are saved for each different combination of query clauses. This can result in a very large number of abstract plans.

If users check the price of a specific title_id using select statements, an abstract plan is saved for each statement. The following two queries each generate an abstract plan:

```
select price from titles where title_id = "T19245"
select price from titles where title_id = "T40007"
```

In addition, there is one plan for each user, that is, if several users check for the title_id “T40007”, a plan is save for each user ID.

If such queries are included in stored procedures, there are two benefits:

- Only only one abstract plan is saved, for example, for the query:

  ```
  select price from titles where title_id =
  @title_id
  ```

- The plan is saved with the user ID of the user who owns the stored procedure, and abstract plan association is made based on the procedure owner's ID.

Using Embedded SQL, the only abstract plan is saved with the host variable:

```
select price from titles
where title_id = :host_var_id
```

CHAPTER 30

# Creating and Using Abstract Plans

This chapter provides an overview of the commands used to capture abstract plans and to associate incoming SQL queries with saved plans. Any user can issue session-level commands to capture and load plans during a session, and a System Administrator can enable server-wide abstract plan capture and association. This chapter also describes how to specify abstract plans using SQL.



## Using *set* commands to capture and associate plans

At the session level, any user can enable and disable capture and use of abstract plans with the set plan dump and set plan load commands. The set plan replace command determines whether existing plans are overwritten by changed plans.

Enabling and disabling abstract plan modes takes effect at the end of the batch in which the command is included (similar to showplan). Therefore, change the mode in a separate batch before you run your queries:

```
set plan dump on
go
/*queries to run*/
go
```

Any set plan commands used in a stored procedure do not affect the procedure in which they are included, but remain in effect after the procedure completes.

## Enabling plan capture mode with *set plan dump*

The set plan dump command activates and deactivates the capture of abstract plans. You can save the plans to the default group, ap_stdout, by using set plan dump with no group name:

```
set plan dump on
```

To start capturing plans in a specific abstract plan group, specify the group name. This example sets the group dev_plans as the capture group:

```
set plan dump dev_plans on
```

The group that you specify must exist before you issue the set command. The system procedure sp_add_qpgroup creates abstract plan groups; only the System Administrator or Database Owner can create an abstract plan group. Once an abstract plan group exists, any user can dump plans to the group. See “Creating a group” on page 696 for information on creating a plan group.

To deactivate the capturing of plans, use:

```
set plan dump off
```

You do not need to specify a group name to end capture mode. Only one abstract plan group can be active for saving or matching abstract plans at any one time. If you are currently saving plans to a group, you must turn off the plan dump mode, and reenable it for the new group, as shown here:

```
set plan dump on /*save to the default group*/
go
/*some queries to be captured */
go
set plan dump off
go
set plan dump dev_plans on
go
/*additional queries*/
go
```

The use of the use database command while set plan dump is in effect disables plan dump mode.

## Associating queries with stored plans

The set plan load command activates and deactivates the association of queries with stored abstract plans.

To start the association mode using the default group, ap_stdin, use the command:

```
set plan load on
```

To enable association mode using another abstract plan group, specify the group name:

```
set plan load test_plans on
```

Only one abstract plan group can be active for plan association at one time. If plan association is active for a group, you must deactivate the current group and start association for the new group, as shown here:

```
set plan load test_plans on
go
/*some queries*/
go
set plan load off
go
set plan load dev_plans on
go
```

The use of the use database command while set plan load is in effect disables plan load mode.

## Using replace mode during plan capture

While plan capture mode is active, you can choose whether to have plans for the same query replace existing plans by enabling or disabling set plan replace. This command activates plan replacement mode:

```
set plan replace on
```

You do not specify a group name with set plan replace; it affects the current active capture group.

To disable plan replacement:

```
set plan replace off
```

The use of the use database command while set plan replace is in effect disables plan replace mode.

### When to use replace mode

When you are capturing plans, and a query has the same query text as an already-saved plan, the existing plan is not replaced unless replace mode is enabled. If you have captured abstract plans for specific queries, and you are making physical changes to the database that affect optimizer choices, you need to replace existing plans for these changes to be saved.

Some actions that might require plan replacement are:

- Adding or dropping indexes, or changing the keys or key ordering in indexes
- Changing the partitioning on a table
- Adding or removing buffer pools
- Changing configuration parameters that affect query plans

For plans to be replaced, plan load mode should not be enabled in most cases. When plan association is active, any plan specifications are used as inputs to the optimizer. For example, if a full query plan includes the prefetch property and an I/O size of 2K, and you have created a 16K pool and want to replace the prefetch specification in the plan, do not enable plan load mode.

You may want to check query plans and replace some abstract plans as data distribution changes in tables, or after rebuilds on indexes, updating statistics, or changing the locking scheme.

## Using dump, load, and replace modes simultaneously

You can have both plan dump and plan load mode active simultaneously, with or without replace mode active.

### Using *dump* and *load* to the same group

If you have enabled dump and load to the same group, without replace mode enabled:

- If a valid plan exists for the query, it is loaded and used to optimize the query.
- If a plan exists that is not valid (say, because an index has been dropped) a new plan is generated and used to optimize the query, but is not saved.

- If the plan is a partial plan, a full plan is generated, but the existing partial plan is not replaced
- If a plan does not exist for the query, a plan is generated and saved.

With replace mode also enabled:

- If a valid plan exists for the query, it is loaded and used to optimize the query.
- If the plan is not valid, a new plan is generated and used to optimize the query, and the old plan is replaced.
- If the plan is a partial plan, a complete plan is generated and used, and the existing partial plan is replaced. The specifications in the partial plan are used as input to the optimizer.
- If a plan does not exist for the query, a plan is generated and saved.

## Using *dump* and *load* to different groups

If you have dump enabled to one group, and load enabled from another group, without replace mode enabled:

- If a valid plan exists for the query in the load group, it is loaded and used. The plan is saved in the dump group, unless a plan for the query already exists in the dump group.
- If the plan in the load group is not valid, a new plan is generated. The new plan is saved in the dump group, unless a plan for the query already exists in the dump group.
- If the plan in the load group is a partial plan, a full plan is generated and saved in the dump group, unless a plan already exists. The specifications in the partial plan are used as input to the optimizer.
- If there is no plan for the query in the load group, the plan is generated and saved in the dump group, unless a plan for the query exists in the dump group.

With replace mode active:

- If a valid plan exists for the query in the load group, it is loaded and used.
- If the plan in the load group is not valid, a new plan is generated and used to optimize the query. The new plan is saved in the dump group.

- If the plan in the load group is a partial plan, a full plan is generated and saved in the dump group. The specifications in the partial plan are used as input to the optimizer.
- If a plan does not exist for the query in the load group, a new plan is generated. The new plan is saved in the dump group.

## *set plan exists check* option

The exists check mode can be used during query plan association to speed performance when users require abstract plans for fewer than 20 queries from an abstract plan group. If a small number of queries require plans to improve their optimization, enabling exists check mode speeds execution of all queries that do not have abstract plans, because they do not check for plans in sysqueryplans.

When set plan load and set exists check are both enabled, the hash keys for up to 20 queries in the load group are cached for the user. If the load group contains more than 20 queries, exists check mode is disabled. Each incoming query is hashed; if its hash key is not stored in the abstract plan cache, then there is no plan for the query and no search is made. This speeds the compilation of all queries that do not have saved plans.

The syntax is:

```
set plan exists check {on | off}
```

You must enable load mode before you enable plan hash-key caching.

A System Administrator can configure server-wide plan hash-key caching with the configuration parameter abstract plan cache. To enable server-wide plan caching, use:

```
sp_configure "abstract plan cache", 1
```

## Using Other *set* options with abstract plans

You can combine other set tuning options with set plan dump and set plan load.

## Using *showplan*

When showplan is turned on, and abstract plan association mode has been enabled with set plan load, showplan prints the plan ID of the matching abstract plan at the beginning of the showplan output for the statement:

```
QUERY PLAN FOR STATEMENT 1 (at line 1).
Optimized using an Abstract Plan (ID : 832005995).
```

If you run queries using the plan clause added to a SQL statement, showplan displays:

```
Optimized using the Abstract Plan in the PLAN clause.
```

## Using *noexec*

You can use noexec mode to capture abstract plans without actually executing the queries. If noexec mode is in effect, queries are optimized and abstract plans are saved, but no query results are returned.

To use noexec mode while capturing abstract plans, execute any needed procedures (such as sp_add_qpgroup) and other set options (such as set plan dump) before enabling noexec mode. The following example shows a typical set of steps:

```
sp_add_qpgroup pubs_dev
go
set plan dump pubs_dev on
go
set noexec on
go
select type, sum(price) from titles group by type
go
```

## Using *forceplan*

If set forceplan on is in effect, and query association is also enabled for the session, forceplan is ignored if a full abstract plan is used to optimize the query. If a partial plan does not completely specify the join order:

- First, the tables in the abstract plan are ordered, as specified.
- The remaining tables are ordered as specified in the from clause.
- The two lists of tables are merged.

# Server-wide abstract plan capture and association Modes

A System Administrator can enable server-wide plan capture, association, and replacement modes with these configuration parameters:

- abstract plan dump – enables dumping to the default abstract plans capture group, ap_stdout.
- abstract plan load – enables loading from the default abstract plans loading group, ap_stdin.
- abstract plan replace – when plan dump mode is also enabled, enables plan replacement.
- abstract plan cache – enables caching of abstract plan hash IDs; abstract plan load must also be enabled. See “set plan exists check option” on page 688 for more information.

By default, these configuration parameters are set to 0, which means that capture and association modes are off. To enable a mode, set the configuration value to 1:

```
sp_configure "abstract plan dump", 1
```

Enabling any of the server-wide abstract plan modes is dynamic; you do not have to reboot the server.

Server-wide capture and association allows the System Administrator to capture all plans for all users on a server. You cannot override he server-wide modes at the session level.

# Creating plans using SQL

You can directly specify the abstract plan for a query by:

- Using the create plan command
- Adding the plan clause to select, insert...select, update, delete and return commands, and to if and while clauses

For information on writing plans, see Chapter 29, “Abstract Query Plan Guide.”

## Using *create plan*

The create plan command specifies the text of a query, and the abstract plan to save for the query.

This example creates an abstract plan:

```
create plan
    "select avg(price) from titles"
"( plan
    ( i_scan type_price_ix titles )
    ( )
)"
```

The plan is saved in the current active plan group. You can also specify the group name:

```
create plan
    "select avg(price) from titles"
"( plan
    ( i_scan type_price_ix titles )
    ( )
)"
into dev_plans
```

If a plan already exists for the specified query in the current plan group, or the plan group that you specify, you must first enable replace mode in order to overwrite the existing plan.

If you want to see the plan ID that is used for a plan you create, create plan can return the ID as a variable. You must declare the variable first. This example returns the plan ID:

```
declare @id int
create plan
    "select avg(price) from titles"
"( plan
    ( i_scan type_price_ix titles )
    ( )
)"
into dev_plans
and set @id
select @id
```

When you use create plan, the query in the plan is not executed. This means that:

- The text of the query is not parsed, so the query is not checked for valid SQL syntax.

- The plans are not checked for valid abstract plan syntax.
- The plans are not checked to determine whether they are compatible with the SQL text.

To guard against errors and problems, you should immediately execute the specified query with showplan enabled.

## Using the *plan* Clause

You can use the plan clause with the following SQL statements to specify the plan to use to execute the query:

- select
- insert...select
- delete
- update
- if
- while
- return

This example specifies the plan to use to execute the query:

```
select avg(price) from titles
     plan
" ( plan
    ( i_scan type_price_ix titles )
    ( )
)"
```

When you specify an abstract plan for a query, the query is executed using the specified plan. If you have showplan enabled, this message is printed:

```
Optimized using the Abstract Plan in the PLAN clause.
```

When you use the plan clause with a query, any errors in the SQL text, the plan syntax, and any mismatches between the plan and the SQL text are reported as errors. For example, this plan omits the empty parentheses that specify the step of returning the aggregate:

```
/* step missing! */
select avg(price) from titles
     plan
" ( plan
```

```
            ( i_scan type_price titles )
        )"
```

It returns the following message:

```
Msg 1005, Level 16, State 1:
Server 'smj', Line 2:
Abstract Plan (AP) : The number of operands of the PLAN operator
in the AP differs from the number of steps needed to compute the
query. The extra items will be ignored. Check the AP syntax and
its correspondence to the query.
```

Plans specified with the plan clause are saved in sysqueryplans only if plan capture is enabled. If a plan for the query already exists in the current capture group, you must enable replace mode in order to replace an existing plan.

CHAPTER 31

# Managing Abstract Plans with System Procedures

This chapter provides an introduction to the basic functionality and use of the system procedures for working with abstract plans. For detailed information on each procedure, see the Adaptive Server Reference Manual.



## System procedures for managing abstract plans

The system procedures for managing abstract plans work on individual plans or on abstract plan groups.

- Managing an abstract plan group
  - sp_add_qpgroup
  - sp_drop_qpgroup
  - sp_help_qpgroup
  - sp_rename_qpgroup
- Finding abstract plans
  - sp_find_qplan
- Managing individual abstract plans
  - sp_help_qplan

    - sp_copy_qplan
    - sp_drop_qplan
    - sp_cmp_qplans
    - sp_set_qplan
- Managing all plans in a group
    - sp_copy_all_qplans
    - sp_cmp_all_qplans
    - sp_drop_all_qplans
- Importing and exporting groups of plans
    - sp_export_qpgroup
    - sp_import_qpgroup

# Managing an abstract plan group

You can use system procedures to create, drop, rename, and provide information about an abstract plan group.

## Creating a group

sp_add_qpgroup creates and names an abstract plan group. Unless you are using the default capture group, ap_stdout, you must create a plan group before you can begin capturing plans. For example, to start saving plans in a group called dev_plans, you must create the group, then issue the set plan dump command, specifying the group name:

```
sp_add_qpgroup dev_plans
set plan dump dev_plans on
/*SQL queries to capture*/
```

Only a System Administrator or Database Owner can add abstract plan groups. Once a group is created, any user can dump or load plans from the group.

## Dropping a group

sp_drop_qpgroup drops an abstract plan group.

The following restrictions apply to sp_drop_qpgroup:

- Only a System Administrator or Database Owner can drop abstract plan groups.
- You cannot drop a group that contains plans. To remove all plans from a group, use sp_drop_all_qplans, specifying the group name.
- You cannot drop the default abstract plan groups ap_stdin and ap_stdout.

This command drops the dev_plans plan group:

```
sp_drop_qpgroup dev_plans
```

## Getting information about a group

sp_help_qpgroup prints information about an abstract plan group, or about all abstract plan groups in a database.

When you use sp_help_qpgroup without a group name, it prints the names of all abstract plan groups, the group IDs, and the number of plans in each group:

```
    sp_help_qpgroup
Query plan groups in database 'pubtune'
 Group                          GID         Plans
 ------------------------------ ----------- -----------
 ap_stdin                                 1           0
 ap_stdout                                2           2
 p_prod                                   4           0
 priv_test                                8           1
 ptest                                    3          51
 ptest2                                   7         189
```

When you use sp_help_qpgroup with a group name, the report provides statistics about plans in the specified group. This example reports on the group ptest2:

```
sp_help_qpgroup ptest2
Query plans group 'ptest2', GID 7

 Total Rows  Total QueryPlans
```

```
 ----------- ----------------
         452              189
sysqueryplans rows consumption, number of query
plans per row count
 Rows        Plans
 ----------- -----------
           5           2
           3          68
           2         119
Query plans that use the most sysqueryplans rows
 Rows        Plan
 ----------- -----------
           5  1932533918
           5  1964534032
 Hashkeys
 -----------
         123
There is no hash key collision in this group.
```

When reporting on an individual group, sp_help_qpgroup reports:

- The total number of abstract plans, and the total number of rows in the sysqueryplans table.
- The number of plans that have multiple rows in sysqueryplans. They are listed in descending order, starting with the plans with the largest number of rows.
- Information about the number of hash keys and hash-key collisions. Abstract plans are associated with queries by a hashing algorithm over the entire query.

When a System Administrator or the Database Owner executes sp_help_qpgroup, the procedure reports on all of the plans in the database or in the specified group. When any other user executes sp_help_qpgroup, it reports only on plans that he or she owns.

sp_help_qpgroup provides several report modes. The report modes are:

| Mode | Information returned |
|---|---|
| full | The number of rows and number of plans in the group, the number of plans that use two or more rows, the number of rows and plan IDs for the longest plans, and number of hash keys, and has- key collision information. This is the default report mode. |
| stats | All of the information from the full report, except hash-key information. |

| Mode | Information returned |
|---|---|
| hash | The number of rows and number of abstract plans in the group, the number of hash keys, and hash-key collision information. |
| list | The number of rows and number of abstract plans in the group, and the following information for each query/plan pair: hash key, plan ID, first few characters of the query, and the first few characters of the plan. |
| queries | The number of rows and number of abstract plans in the group, and the following information for each query: hash key, plan ID, first few characters of the query. |
| plans | The number of rows and number of abstract plans in the group, and the following information for each plan: hash key, plan ID, first few characters of the plan. |
| counts | The number of rows and number of abstract plans in the group, and the following information for each plan: number of rows, number of characters, hash key, plan ID, first few characters of the query. |

This example shows the output for the counts mode:

```
                sp_help_qpgroup ptest1, counts
Query plans group 'ptest1', GID 3

 Total Rows  Total QueryPlans
 ----------- ----------------
         48               19

Query plans in this group

Rows  Chars    hashkey    id         query
----- --------- ----------- ----------- ----------------------------
    3      623  1801454852   876530156 select title from titles ...
    3      576   476063777   700529529 select au_lname, au_fname...
    3      513   444226348   652529358 select au1.au_lname, au1....
    3      470   792078608   716529586 select au_lname, au_fname...
    3      430   789259291   684529472 select au1.au_lname, au1....
    3      425  1929666826   668529415 select au_lname, au_fname...
    3      421   169283426   860530099 select title from titles ...
    3      382   571605257   524528902 select pub_name from publ...
    3      355   845230887   764529757 delete salesdetail where ...
    3      347   846937663   796529871 delete salesdetail where ...
    2      379  1400470361   732529643 update titles set price =...
```

## Renaming a group

A System Administrator or Database Owner can rename an abstract plan group with sp_rename_qpgroup. This example changes the name of the group from dev_plans to prod_plans:

```
sp_rename_qpgroup dev_plans, prod_plans
```

The new group name cannot be the name of an existing group.

# Finding abstract plans

sp_find_qplan searches both the query text and the plan text to find plans that match a given pattern.

This example finds all plans where the query includes the string “from titles”:

```
sp_find_qplan "%from titles%"
```

This example searches for all abstract plans that perform a table scan:

```
sp_find_qplan "%t_scan%"
```

When a System Administrator or Database Owner executes sp_find_qplan, the procedure examines and reports on plans owned by all users. When other users execute the procedure, it searches and reports on only plans that they own.

If you want to search just one abstract plan group, specify the group name with sp_find_qplan. This example searches only the test_plans group, finding all plans that use a particular index:

```
sp_find_qplan "%i_scan title_id_ix%", test_plans
```

For each matching plan, sp_find_qplan prints the group ID, plan ID, query text, and abstract plan text.

# Managing individual abstract plans

You can use system procedures to print the query and text of individual plans, to copy, drop, or compare individual plans, or to change the plan associated with a particular query.

## Viewing a plan

sp_help_qplan reports on individual abstract plans. It provides three types of reports that you can specify: brief, full, and list. The brief report prints only the first 78 characters of the query and plan; use full to see the entire query and plan, or list to display only the first 20 characters of the query and plan.

This example prints the default brief report:

```
            sp_help_qplan 588529130
 gid         hashkey     id
 ----------- ----------- -----------
           8  1460604254   588529130
 query
 ------------------------------------------------------------------
 select min(price) from titles
 plan
-------------------------------------------------------------------
 ( plan
    ( i_scan type_price titles )
    ( )
)
( prop titles
    ( parallel ...
```

A System Administrator or Database Owner can use sp_help_qplan to report on any plan in the database. Other users can only view the plans that they own.

sp_help_qpgroup reports on all plans in a group. For more information see “Getting information about a group” on page 697.

## Copying a plan to another group

sp_copy_qplan copies an abstract plan from one group to another existing group. This example copies the plan with plan ID 316528161 from its current group to the prod_plans group:

```
sp_copy_qplan 316528161, prod_plans
```

sp_copy_qplan checks to make sure that the query does not already exist in the destination group. If a possible conflict exists, it runs sp_cmp_qplans to check plans in the destination group. In addition to the message printed by sp_cmp_qplans, sp_copy_qplan prints messages when:

- The query and plan you are trying to copy already exists in the destination group
- Another plan in the group has the same user ID and hash key
- Another plan in the group has the same hash key, but the queries are different

If there is a hash-key collision, the plan is copied. If the plan already exists in the destination group or if it would give an association key collision, the plan is not copied. The messages printed by sp_copy_qplan contain the plan ID of the plan in the destination group, so you can use sp_help_qplan to check the query and plan.

A System Administrator or the Database Owner can copy any abstract plan. Other users can copy only plans that they own. The original plan and group are not affected by sp_copy_qplan. The copied plan is assigned a new plan ID, the ID of the destination group, and the user ID of the user who ran the query that generated the plan.

## Dropping an individual abstract plan

sp_drop_qplan drops individual abstract plans. This example drops the specified plan:

```
sp_drop_qplan 588529130
```

A System Administrator or Database Owner can drop any abstract plan in the database. Other users can drop only plans that they own.

To find abstract plan IDs, use sp_find_qplan to search for plans using a pattern from the query or plan, or sp_help_qpgroup to list the plans in a group.

## Comparing two abstract plans

Given two plan IDs, sp_cmp_qplans compares two abstract plans and the associated queries. For example:

```
sp_cmp_qplans 588529130, 1932533918
```

sp_cmp_qplans prints one message reporting the comparison of the query, and a second message about the plan, as follows:

- For the two queries, one of:
  - The queries are the same.
  - The queries are different.
  - The queries are different but have the same hash key.
- For the plans:
  - The query plans are the same.
  - The query plans are different.

This example compares two plans where the queries and plans both match:

```
sp_cmp_qplans 411252620, 1383780087
The queries are the same.
The query plans are the same.
```

This example compares two plans where the queries match, but the plans are different:

```
sp_cmp_qplans 2091258605, 647777465
The queries are the same.
The query plans are different.
```

sp_cmp_qplans returns a status value showing the results of the comparison. The status values are shown in Table 31-1

***Table 31-1: Return status values for sp_cmp_qplans***

| Return value | Meaning |
|---|---|
| 0 | The query text and abstract plans are the same. |
| +1 | The queries and hash keys are different. |
| +2 | The queries are different, but the hash keys are the same. |
| +10 | The abstract plans are different. |
| 100 | One or both of the plan IDs does not exist. |

A System Administrator or Database Owner can compare any two abstract plans in the database. Other users can compare only plans that they own.

## Changing an existing plan

sp_set_qplan changes the abstract plan for an existing plan ID without changing the ID or the query text. It can be used only when the plan text is 255 or fewer characters.

```
sp_set_qplan 588529130, "( i_scan title_ix titles)"
```

A System Administrator or Database Owner can change the abstract plan for any saved query. Other users can modify only plans that they own.

When you execute sp_set_qplan, the abstract plan is not checked against the query text to determine whether the new plan is valid for the query, or whether the tables and indexes exist. To test the validity of the plan, execute the associated query.

You can also use create plan and the plan clause to specify the abstract plan for a query. See “Creating plans using SQL” on page 690.

# Managing all plans in a group

These system procedures help manage groups of plans:

- sp_copy_all_qplans
- sp_cmp_all_qplans
- sp_drop_all_qplans

## Copying all plans in a group

sp_copy_all_qplans copies all of the plans in one abstract plan group to another group. This example copies all of the plans from the test_plans group to the helpful_plans group:

```
sp_copy_all_qplans test_plans, helpful_plans
```

The helpful_plans group must exist before you execute sp_copy_all_qplans. It can contain other plans.

sp_copy_all_qplans copies each plan in the group by executing sp_copy_qplan, so copying a plan may fail for the same reasons that sp_copy_qplan might fail. See “Comparing two abstract plans” on page 703.

Each plan is copied as a separate transaction, and failure to copy any single plan does not cause sp_copy_all_qplans to fail. If sp_copy_all_qplans fails for any reason, and has to be restarted, you see a set of messages for the plans that have already been successfully copied, telling you that they exist in the destination group.

A new plan ID is assigned to each copied plan. The copied plans have the original user's ID. To copy abstract plans and assign new user IDs, you must use sp_export_qpgroup and sp_import_qpgroup. See "Importing and exporting groups of plans" on page 708.

A System Administrator or Database Owner can copy all plans in the database. Other users can copy only plans that they own.

## Comparing all plans in a group

sp_cmp_all_qplans compares all abstract plans in two groups and reports:

- The number of plans that are the same in both groups
- The number of plans that have the same association key, but different abstract plans
- The number of plans that are present in one group, but not the other

This example compares the plans in ap_stdout and ap_stdin:

```
    sp_cmp_all_qplans ap_stdout, ap_stdin
If the two query plans groups are large, this might take some
time.
Query plans that are the same
 count
 -----------
         338
Different query plans that have the same association key

 count
 -----------
          25
Query plans present only in group 'ap_stdout' :

 count
 -----------
           0
Query plans present only in group 'ap_stdin' :
```

```
count
-----------
          1
```

With the additional specification of a report-mode parameter, sp_cmp_all_qplans provides detailed information, including the IDs, queries, and abstract plans of the queries in the groups. The mode parameter lets you get the detailed information for all plans, or just those with specific types of differences.Table 31-2 shows the report modes and what type of information is reported for each mode.

***Table 31-2: Report modes for sp_cmp_all_qplans***

| Mode | Reported information |
|---|---|
| counts | The counts of: plans that are the same, plans that have the same association key, but different groups, and plans that exist in one group, but not the other. This is the default report mode. |
| brief | The information provided by counts, plus the IDs of the abstract plans in each group where the plans are different, but the association key is the same, and the IDs of plans that are in one group, but not in the other. |
| same | All counts, plus the IDs, queries, and plans for all abstract plans where the queries and plans match. |
| diff | All counts, plus the IDs, queries, and plans for all abstract plans where the queries and plans are different. |
| first | All counts, plus the IDs, queries, and plans for all abstract plans that are in the first plan group, but not in the second plan group. |
| second | All counts, plus the IDs, queries, and plans for all abstract plans that are in the second plan group, but not in the first plan group. |
| offending | All counts, plus the IDs, queries, and plans for all abstract plans that have different association keys or that do not exist in both groups. This is the combination of the diff, first, and second modes. |
| full | All counts, plus the IDs, queries, and plans for all abstract plans. This is the combination of same and offending modes. |

This example shows the brief report mode:

```
       sp_cmp_all_qplans ptest1, ptest2, brief
If the two query plans groups are large, this might take
some time.
Query plans that are the same
 count
 -----------
          39
Different query plans that have the same association key
```

```
 count
 -----------
           4

    ptest1    ptest2

 id1         id2
 ----------- -----------
   764529757  1580532664
   780529814  1596532721
   796529871  1612532778
   908530270  1724533177
Query plans present only in group 'ptest1' :

 count
 -----------
           3


 id
 -----------
   524528902
  1292531638
  1308531695

Query plans present only in group 'ptest2' :

 count
 -----------
           1


 id
 -----------
  2108534545
```

## Dropping all abstract plans in a group

sp_drop_all_qplans drops all abstract plans in a group. This example drops all abstract plans in the dev_plans group:

```
sp_drop_all_qplans dev_plans
```

When a System Administrator or the Database Owner executes sp_drop_all_qplans, all plans belonging to all users are dropped from the specified group. When another user executes this procedure, it affects only the plans owned by that users.

# Importing and exporting groups of plans

sp_export_qpgroup and sp_import_qpgroup copy groups of plans between sysqueryplans and a user table. This allows a System Administrator or Database Owner to:

- Copy abstract plans from one database to another on the same server
- Create a table that can be copied out of the current server with bcp, and copied into another server
- Assign different user IDs to existing plans in the same database

## Exporting plans to a user table

sp_export_qpgroup copies all plans for a specific user from an abstract plan group to a user table. This example copies plans owned by the Database Owner (dbo) from the fast_plans group, creating a table called transfer:

```
sp_export_qpgroup dbo, fast_plans, transfer
```

sp_export_qpgroup uses select...into to create a table with the same columns and datatypes as sysqueryplans. If you do not have the select into/bulkcopy/pllsort option enabled in the database, you can specify the name of another database. This command creates the export table in tempdb:

```
sp_export_qpgroup mary, ap_stdout, "tempdb..mplans"
```

The table can be copied out using bcp, and copied into a table on another server. The plans can also be imported to sysqueryplans in another database on the same server, or the plans can be imported into sysqueryplans in the same database, with a different group name or user ID.

## Importing plans from a user table

sp_import_qpgroup copies plans from tables created by sp_export_qpgroup into a group in sysqueryplans. This example copies the plans from the table tempdb..mplans into ap_stdin, assigning the user ID for the Database Owner:

```
sp_import_qpgroup "tempdb..mplans", dbo, ap_stdin
```

You cannot copy plans into a group that already contains plans for the specified user.

CHAPTER 32

# Abstract Plan Language Reference

This chapter describes the operators and other language elements in the abstract plan language.



## Keywords

The following words are keywords in the abstract query plan language. They are not reserved words, and do not conflict with the names of tables or indexes used in abstract plans. For example, a table or index may be named hints.

## Operands

The following operands are used in the abstract plan syntax statements:

***Table 32-1: Identifiers used***

| Identifier | Describes |
|---|---|
| *table_name* | The name of a base table, that is, a user or system table |
| *correlation_name* | The correlation name specified for a table in a query |
| *derived_table* | A table that results from the scan of a stored table |
| *stored_table* | A base table or a worktable |
| *worktable_name* | The name of a worktable |
| *view_name* | The name of a view |
| *index_name* | The name of an index |
| *subquery_id* | An integer identifying the order of the subqueries in the query |

*table_name* and *view_name* can be specified using the notation *database.owner.object_name*.

## Derived tables

A derived table is a result of access to a stored table during query execution. It can be:

- The result set generated by the query
- An intermediate result during query execution; that is, the result of the join of the first two tables in the join order, which is then joined with a third table

Derived tables result from one of the scan operators that specify the access method: scan, i_scan, or t_scan, for example, (i_scan title_id_ix titles).

# Schema for examples

To simplify the sample abstract plan examples, the following tables are used in this section:

```
create table t1 (c11 int, c12 int)
create table t2 (c21 int, c22 int)
create table t3 (c31 int, c32 int)
```

The following indexes are used:

```
create index i_c11 on t1(c11)
```

```
create index i_c12 on t1(c12)
create index i_c11_c12 on t1(c11, c12)
create index i_c21 on t2(c21)
create index i_c22 on t2(c22)
create index i_c31 on t3(c31)
create index i_c32 on t3(c32)
```

# g_join

Description

Specifies the join of two or more derived tables without specifying the join type (nested-loop or sort-merge).

Syntax

( g_join *derived_table1 derived_table2*
)

( g_join ( *derived_table1* )
( *derived_table2* )
...
( *derived_tableN* )
)

Parameters

*derived_table1*...*derived_tableN*
are the derived tables to be united.

Return value

A derived table that is the join of the specified derived tables.

Examples

**Example 1**

```
select *
from t1, t2
where c21 = 0
and c22 = c12
```

```
( g_join
    ( i_scan i_c21 t2 )
    ( i_scan i_c12 t1 )
)
```

Table t2 is the outer table, and t1 the inner table in the join order.

**Example 2**

```
select *
from t1, t2, t3
where c21 = 0
and c22 = c12
```

```
and c11 = c31


( g_join
    ( i_scan i_c21 t2 )
    ( i_scan i_c12 t1 )
    ( i_scan i_c31 t3 )
)
```

Table t2 is joined with t1, and the derived table is joined with t3.

Usage

- The g_join operator is a generic logical operator that describes all binary joins (inner join, outer join, or existence join).
- The g_join operator is never used in generated plans; nl_g_join and m_g_join operators indicate the join type used.
- The optimizer chooses between a nested-loop join and a sort-merge join when the g_join operator is used. To specify a sort-merge join, use m_g_join. To specify a nested-loop join, use nl_g_join.
- The syntax provides a shorthand method of described a join involving multiple tables. This syntax:

```
( g_join
        ( scan t1)
        ( scan t2)
        ( scan t3)
        ...
        ( scan tN-1)
        ( scan tN)
)
```

is shorthand for:

```
( g_join
        ( g_join
                ...
                    ( g_join
                            (g_join
                                    ( scan t1)
                                    ( scan t2)
                            )
                            ( scan t3)
                    )
                ...
                ( scan tN-1)
        )
        ( scan tN)
```

```
)
```

- If g_join is used to specify the join order for some, but not all, of the tables in a query, the optimizer uses the join order specified, but may insert other tables between the g_join operands. For example, for this query:

  ```
  select *
      from t1, t2, t3
      where ...
  ```

  the following partial abstract plan describes only the join order of t1 and t2:

  ```
  ( g_join
          ( scan t2)
          ( scan t1)
  )
  ```

  The optimizer can choose any of the three join orders: t3-t2-t1, t2-t3-t1 or t2-t1-t3.

- The tables are joined in the order specified in the g_join clause.
- If set forceplan on is in effect, and query association is also enabled for the session, forceplan is ignored if a full abstract plan is used to optimize the query. If a partial plan does not completely specify the join order:
  - First, the tables in the abstract plan are ordered as specified.
  - The remaining tables are ordered as specified in the from clause.
  - The two lists of tables are merged.

See also

m_g_join, nl_g_join

# hints

Description

Introduces and groups items in a partial abstract plan.

Syntax

```
( hints ( derived_table )
    ...
)
```

Parameters

*derived_table*
is one or more expressions that generate a derived table.

Return value

A derived table.

Examples

```
select *
from t1, t2
where c12 = c21
        and c11 > 0
        and c22 < 1000


( hints
    ( g_join
        ( t_scan t2 )
        ( i_scan () t1 )
    )
)
```

Specifies a partial plan, including a table scan on t2, the use of some index on t1, and the join order t1-t2. The index choice for t1 and the type of join (nested-loop or sort-merge) is left to the optimizer.

Usage

- The specified hints are used during query optimization.
- The hints operator appears as the root of a partial abstract plan that includes multiple steps. If a partial plan contains only one expression, hints is optional.
- The hints operator does not appear in plans generated by the optimizer; these are always full plans.
- Hints can be associated with queries:
  - By changing the plan for an existing query with sp_set_qplan.
  - By specifying the plan for a query with the plan clause. To save the query and hints, set plan dump must be enabled.
  - By using the create plan command.
- When hints are specified in the plan clause for a SQL statement, the plans are checked to be sure they are valid. When hints are specified using sp_set_qplan, plans are not checked before being saved.

# i_scan

Description

Specifies an index scan of a base table.

Syntax

( i_scan *index_name base_table* )

( i_scan () *base_table* )

Parameters

*index_name*

is the name or index ID of the index to use for an index scan of the specified stored table. Use of empty parentheses specify that an index scan (rather than table scan) is to be performed, but leaves the choice of index to the optimizer.

*base_table*

is the name of the base table to be scanned.

Return value

A derived table produced by a scan of the base table.

Examples

**Example 1**

```
select * from t1 where c11 = 0


( i_scan i_c11 t1 )
```

Specifies the use of index i_c11 for a scan of t1.

**Example 2**

```
select *
    from t1, t2
    where c11 = 0
        and c22 = 1000
        and c12 = c21


( g_join
        ( scan t2 )
        ( i_scan () t1 )
)
```

Specifies a partial plan, indicating the join order, but allowing the optimizer to choose the access method for t2, and the index for t1.

```
select * from t1 where c12 = 0


( i_scan 2 t1 )
```

Identifies the index on t1 by index ID, rather than by name.

Usage

- The index is used to scan the table, or, if no index is specified, an index is used rather than a table scan.

- Use of empty parentheses after the i_scan operator allows the optimizer to choose the index or to perform a table scan if no index exists on the table.
- When the i_scan operator is specified, a covering index scan is always performed when all of the required columns are included in the index. No abstract plan specification is needed to describe a covering index scan.
- Use of the i_scan operator suppresses the choice of the reformatting strategy and the OR strategy, even if the specified index does not exist. The optimizer chooses another useful index and an advisory message is printed. If no index is specified for i_scan, or if no indexes exist, a table scan is performed, and an advisory message is printed.
- Although specifying an index using the index ID is valid in abstract query plans, using an index ID is not recommended. If indexes are dropped and re-created in a different order, plans become invalid or perform suboptimally.

See also

scan, t_scan

# in

Description

Identifies the location of a table that is specified in a subquery or view.

Syntax

( in ( [ subq *subquery_id* | view *view_name* ] )
)

Parameters

subq *subquery_id*
is an integer identifying a subquery. In abstract plans, subquery numbering is based on the order of the leading open parentheses for the subqueries in a query.

view *view_name*
is the name of a view. The specification of database and owner name in the abstract plan must match the usage in the query in order for plan association to be performed.

Examples

**Example 1**

```
create view v1 as
select * from t1

select * from v1
```

```
( t_scan ( table t1 ( in ( view v1 ) ) ) )
```

Identifies the view in which table t1 is used.

**Example 2**

```
select *
from t2
where c21
in (select c12 from t1)


( g_join
    ( t_scan t2 )
    ( t_scan ( table t1 ( in ( subq 1 ) ) ) )
)
```

Identifies the scan of table *t1* in subquery 1.

**Example 3**

```
create view v9
as
select *
from t1
where c11 in (select c21 from t2)


create view v10
as
select * from v9
where c11 in (select c11 from v9)


select * from v10, t3
where c11 in
        (select c11 from v10 where c12 = t3.c31)


( g_join
( t_scan t3 )
( i_scan i_c21 ( table t2 ( in ( subq 1 ) ( view v9 ) ( view v10 ))))
( i_scan i_c11 ( table t1 ( in ( view v9 ) ( view v10 ))))
( i_scan i_c11 ( table t1 ( in ( view v9 ) ( view v10 ) ( subq 1 ))))
```

```
( i_scan i_c11 ( table t1 ( in ( view v9 ) ( subq 1 ) ( view v10 ))))
( i_scan i_c21 ( table t2 ( in ( subq 1 ) ( view v9 ) ( subq 1 ) ( view v10 ))))
( i_scan i_c11 ( table t1 ( in ( view v9 ) ( subq 1 ) ( view v10 ) ( subq 1 ))))
( i_scan i_c21 ( table t2 ( in ( subq 1 ) ( view v9 ) ( view v10 ) ( subq 1 ))))
( i_scan i_c21 ( table t2(in( subq 1 )( view v9 )( subq 1 )( view v10 ) (
subq 1))))
)
```

An example of multiple nesting of views and subqueries.

Usage

- Identifies the occurrence of a table in view or subqueryof the SQL query.
- The in list has the innermost items to the left, near the table's name (itself the deeply nested item), and the outermost items (the ones occurring in the top level query) to the right. For example, the qualification:

  ```
  (table t2 (in (subq 1) (view v9) (subq 1) (view
  v10) (subq 1) ) )
  ```

  can be read in either direction:

  - Reading left to right, starting from the table: the base table t2 as scanned in the first subquery of view v9 , which occurs in the first subquery of view v10 , which occurs in the first subquery of the main query
  - Reading from right to left, that is, starting from the main query: in the main query there's a first subquery, that scans the view v10 , that contains a first subquery that scans the view v9 , that contains a first subquery that scans the base table t2

See also

nested, subq, table, view

# lru

Description

Specifies LRU cache strategy for the scan of a stored table.

Syntax

```
( prop table_name
      ( lru )
)
```

Parameters

*table_name*
is the table to which the property is to be applied.

Examples

```
select * from t1
```

```
( prop t1
        ( lru)
)
```

Specifies the use of LRU cache strategy for the scan of t1.

Usage

- LRU strategy is used in the resulting query plan.
- Partial plans can specify scan properties without specifying other portions of the query plan.
- Full query plans always include all scan properties.

See also mru, prop

# m_g_join

Description Specifies a merge join of two derived tables.

Syntax

```
( m_g_join (
    ( derived_table1 )
    ( derived_table2 )
)
```

Parameters

*derived_table1...derived_tableN*
are the derived tables to be united. *derived_table1* is always the outer table and *derived_table2* is the inner table

Return value A derived table that is the join of the specified derived tables.

Examples **Example 1**

```
select t1.c11, t2.c21
    from t1, t2, t3
    where t1.c11 = t2.c21
        and t1.c11 = t3.c31


( nl_g_join
        ( m_g_join
            ( i_scan i_c31 t3 )
            ( i_scan i_c11 t1 )
        )
        ( t_scan t2 )
)
```

Specifies a right-merge join of tables t1 and t3, followed by a nested-loop join with table t2.

**Example 2**

```
select * from t1, t2, t3
where t1.c11 = t2.c21 and t1.c11 = t3.c31
and t2.c22 =7


( nl_g_join
    ( m_g_join
        ( i_scan i_c21 t2 )
        ( i_scan i_c11 t1 )
    )
    ( i_scan i_c31 t3 )
)
```

Specifies a full-merge join of tables t2 (outer) and t1 (inner), followed in the join order by a nested-loop join with t3.

**Example 3**

```
select c11, c22, c32
from t1, t2, t3
where t1.c11 = t2.c21
and t2.c22 = t3.c32


( m_g_join
        (nl_g_join
            (i_scan i_c11 t1)
            (i_scan i_c12 t2)
        )
    (i_scan i_c32_ix t3)
)
```

Specifies a nested-loop join of t1 and t2, followed by a merge join with t3.

Usage

- The tables are joined in the order specified in the m_g_join clause.
- The sort step and worktable required to process sort-merge join queries are not represented in abstract plans.
- If the m_g_join operator is used to specify a join that cannot be performed as a merge join, the specification is silently ignored.

See also

g_join, nl_g_join

# mru

Description

Specifies MRU cache strategy for the scan of a stored table.

Syntax

```
( prop table_name
      ( mru)
)
```

Parameters

*table_name*
is the table to which the property is to be applied.

Examples

```
select * from t1


( prop t1
        ( mru )
)
```

Specifies the use of MRU cache strategy for the table.

Usage

- MRU strategy is specified in the resulting query plan
- Partial plans can specify scan properties without specifying other portions of the query plan.
- Generated query plans always include all scan properties.
- If sp_cachestrategy has been used to disable MRU replacement for a table or index, and the query plan specifies MRU, the specification in the abstract plan is silently ignored.

See also

lru, prop

# nested

Description

Describes the nesting of subqueries on a derived table.

Syntax

```
( nested
      ( derived_table )
          ( subquery_specification )
)
```

Parameters

*derived_table*
is the derived table over which to nest the specified subquery.

*subquery_specification*
is the subquery to nest over *derived_table*

Return value

A derived table.

Examples

**Example 1**

```
select c11 from t1
where c12 =
        (select c21 from t2 where c22 = t1.c11)
```

```
( nested
        ( t_scan t1 )
        ( subq 1
                ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
        )
)
```

A single nested subquery.

**Example 2**

```
select c11 from t1
where c12 =
        (select c21 from t2 where c22 = t1.c11)
    and c12 =
        (select c31 from t3 where c32 = t1.c11)
```

```
( nested
        ( nested
                ( t_scan t1 )
                ( subq 1
                        ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
                )
        )
        ( subq 2
                ( t_scan ( table t3 ( in ( subq 2 ) ) ) )
        )
)
```

The two subqueries are both nested in the main query.

**Example 3**

```
select c11 from t1
where c12 =
    (select c21 from t2 where c22 =
        (select c31 from t3 where c32 = t1.c11))
```

```
( nested
        ( t_scan t1 )
        ( subq 1
                ( nested
                        ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
                        ( subq 2
                              ( t_scan ( table t3 ( in ( subq 2 ) ) ) )
                        )
                )
        )
)
```

A level 2 subquery nested into a level 1 subquery nested in the main query.

Usage

- The subquery is executed at the specified attachment point in the query plan.
- Materialized and flattened subqueries do not appear under a nested operator. See subq on subq on page 734 for examples.

See also

in, subq

# nl_g_join

Description

Specifies a nested-loop join of two or more derived tables.

Syntax

```
( nl_g_join     ( derived_table1 )
                ( derived_table2 )
    ...
                ( derived_tableN )
)
```

Parameters

*derived_table1...derived_tableN*
are the derived tables to be united.

Return value

A derived table that is the join of the specified derived tables.

Examples

**Example 1**

```
select *
from t1, t2
where c21 = 0
and c22 = c12


( nl_g_join
    ( i_scan i_c21 t2 )
```

```
        ( i_scan i_c12 t1 )
    )
```

Table t2 is the outer table, and t1 the inner table in the join order.

**Example 2**

```
select *
from t1, t2, t3
where c21 = 0
and c22 = c12
and c11 = c31


( nl_g_join
    ( i_scan i_c21 t2 )
    ( i_scan i_c12 t1 )
    ( i_scan i_c31 t3 )
)
```

Table t2 is joined with t1, and the derived table is joined with t3.

Usage

- The tables are joined in the order specified in the nl_g_join clause
- The nl_g_join operator is a generic logical operator that describes all binary joins (inner join, outer join, or semijoin). The joins are performed using the nested-loops query execution method.

See also

g_join, m_g_join

# parallel

Description

Specifies the degree of parallelism for the scan of a stored table.

Syntax

```
( prop table_name
      ( parallel degree )
)
```

Parameters

*table_name*
is the table to which the property is to be applied.

*degree*
is the degree of parallelism to use for the scan.

Examples

```
select * from t1
```

```
(prop t1
        ( parallel 5 )
)
```

Specifies that 5 worker processes should be used for the scan of the t1 table.

Usage

- The scan is performed using the specified number of worker processes, if available.
- Partial plans can specify scan properties without specifying other portions of the query plan.
- If a saved plan specifies the use of a number of worker processes, but session-level or server-level values are different when the query is executed:
  - If the plan specifies more worker processes than permitted by the current settings, the current settings are used or the query is executed using a serial plan.
  - If the plan specifies fewer worker processes than permitted by the current settings, the values in the plan are used.

  These changes to the query plan are performed transparently to the user, so no warning messages are issued.

See also

prop

# plan

Description

Provides a mechanism for grouping the query plan steps of multi-step queries, such as queries requiring worktables, and queries computing aggregate values.

Syntax

```
(plan
      query_step1
      ...
      query_stepN
)
```

Parameters

*query_step1...query_stepN* –
specify the abstract plan steps for the execution of each step in the query.

Return value

A derived table.

Examples

**Example 1**

```
select max(c11) from t1
group by c12


( plan
    ( store Worktab1
        ( t_scan t1 )
    )
    ( t_scan ( work_t Worktab1 ) )
)
```

Returns a vector aggregate. The first operand of the plan operator creates Worktab1 and specifies a table scan of the base table. The second operand scans the worktable to return the results.

**Example 2**

```
select max(c11) from t1


( plan
    ( t_scan t1 )
    ( )
)
```

Returns a scalar aggregate. The last derived table is empty, because scalar aggregates accumulate the result value in an internal variable rather than a worktable.

**Example 3**

```
select *
from t1
where c11 = (select count(*) from t2)


( plan
    ( i_scan i_c21 (table t2 ( in_subq 1) ) )
    ( i_scan i_c11 t1 )
)
```

Specifies the execution of a materialized subquery.

**Example 4**

```
create view v3
as
```

```
select distinct * from t3
```

```
select * from t1, v3
where c11 = c31
```

```
( plan
    ( store Worktab1
        ( t_scan (table t3 (in_view v3 ) ) )
    )
    ( nl_g_join
        ( t_scan t1 )
        ( t_scan ( work_t Worktab1 ) )
    )
)
```

Specifies the execution of a materialized view.

Usage

- Tables are accessed in the order specified, with the specified access methods.
- The plan operator is required for multistep queries, including:
  - Queries that generate worktables, such as queries that perform sorts and those that compute vector aggregates
  - Queries that compute scalar aggregates
  - Queries that include materialized subqueries
- An abstract plan for a query that requires multiple execution steps must include operands for each step in query execution if it begins with the plan keyword. Use the hints operator to introduce partial plans.

See also

hints

# prefetch

Description

Specifies the I/O size to use for the scan of a stored table.

Syntax

```
( prop table_name
      ( prefetch size )
)
```

Parameters

*table_name*
is the table to which the property is to be applied.

*size*
is a valid I/O size: 2, 4, 8 or 16.

Examples

```
select * from t1


( prop t1
    (prefetch 16 )
)
```

16K I/O size is used for the scan of t1.

Usage

- The specified I/O size is used in the resultant query plan if a pool of that size exists in the cache used by the table.
- Partial plans can specify scan properties without specifying other portions of the query plan.
- If large I/O specifications in a saved plan do not match current pool configuration or other options:
  - If the plan specifies 16K I/O, and the 16K pool does not exist, the next largest available I/O size is used.
  - If session or server-level options have made large I/O unavailable for the query (set prefetch for the session, or sp_cachestrategy for the table), 2K I/O is used.
- If you save plans that specify only 2K I/O for the scan properties, and later create large I/O pools, enable replace mode to save the new plans if you want these plans to use larger I/O sizes.

See also

prop

# prop

Description

Specifies properties to use for the scan of a stored table.

Syntax

( prop *table_name*
    ( *property_specification* ) ...
)

*property_specification*:

( prefetch *size* )
( lru | mru )
( parallel *degree* )

Parameters

*table_name*
is the table to which the property is to be applied.

Examples

```
select * from t1


( t_scan t1 )
( prop t1
    ( parallel 1 )
    ( prefetch 16 )
    ( lru )
)
```

Shows the property values used by the scan of t1.

Usage

- The specified properties are used for the scan of the table
- Partial plans can specify scan properties without specifying other portions of the query plan.
- Generated plans include the parallel, prefetch, and cache strategy properties used for each table in the query.

See also

lru, mru, parallel, prefetch

## scan

Description

Specifies the scan of a stored table, without specifying the type of scan.

Syntax

( scan *stored_table* )

Parameters

*stored_table*
is the name of the stored table to be scanned. It can be a base table or worktable.

Return value

A derived table produced by the scan of the stored table.

Examples

**Example 1**

```
select * from t1 where c11 > 10


( scan t1 )
```

Specifies a scan of t1, leaving the optimizer to choose whether to perform a table scan or index scan.

**Example 2**

```
select *
    from t1, t2
    where c11 = 0
        and c22 = 1000
        and c12 = c21


( nl_g_join
        ( scan t2 )
        ( i_scan i_c22 t1 )
)
```

Specifies a partial plan, indicating the join order, but allowing the optimizer to choose the access method for t2.

Usage

- The optimizer chooses the access method for the stored table.
- The scan operator is used when the choice of the type of scan should be left to the optimizer. The resulting access method can be one of the following:
  - A full table scan
  - An index scan, with access to data pages
  - A covering index scan, with no access to data pages
  - A RID scan, used for the OR strategy
- For an example of an abstract plan that specifies the reformatting strategy, see store.

See also

i_scan, store, t_scan

# store

Description

Stores the results of a scan in a worktable.

Syntax

( store *worktable_name*
    ( [scan | i_scan | t_scan ] *table_name* )
)

Parameters

*worktable_name*
is the name of the worktable to be created.

*table_name*
is the name of the base table to be scanned.

Return value

A worktable that is the result of the scan.

Examples

```
select c12, max(c11) from t1
        group by c12


( plan
    ( store Worktab1
        ( t_scan t1 )
    )
    ( t_scan ( work_t Worktab1 ) )
)
```

Specifies the two-step process of selecting the vector aggregate into a worktable, then selecting the results of the worktable.

Usage

- The specified table is scanned, and the result is stored in a worktable
- The legal places for a store operator in an abstract plan are:
  - Under a plan or union operator, where the store operator signifies a preprocessing step resulting in a worktable
  - Under a scan operator (but not under an i_scan or t_scan operator)
- During plan capture mode, worktables are identified as *Worktab1*, *Worktab2*, and so on. For manually entered plans, any naming convention can be used.
- The use of the reformatting strategy can be described in an abstract plan using the scan (store ( )) combination of operators. For example, if t2 has no indexes and is very large, the abstract plan below indicates that t2 should be scanned once, via a table scan, with the results stored in a worktable:

```
select *
from t1, t2
where c11 > 0
        and c12 = c21
        and c22 between 0 and 10000
( nl_g_join
        (i_scan i_c11 t1)
```

```
                    ( scan (store (t_scan t2 )))
        )
```

See also scan

# subq

Description Identifies a subquery.

Syntax
```
( subq subquery_id
      )
```

Parameters

*subquery_id*
is an integer identifying the subquery. In abstract plans, subquery numbering is based on the order of the leading parenthesis for the subqueries in a query.

Examples

**Example 1**

```
select c11 from t1
where c12 =
        (select c21 from t2 where c22 = t1.c11)


( nested
        ( t_scan t1 )
        ( subq 1
            ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
        )
)
```

A single nested subquery.

**Example 2**

```
select c11 from t1
where c12 =
        (select c21 from t2 where c22 = t1.c11)
    and c12 =
        (select c31 from t3 where c32 = t1.c11)


( nested
        ( nested
                ( t_scan t1 )
                ( subq 1
```

```
                                ( t_scan ( table t2 ( in (
subq 1 ) ) ) )
                        )
                )
                ( subq 2
                    ( t_scan ( table t3 ( in ( subq 2 ) ) ) )
                )
)
```

The two subqueries are both nested in the main query.

**Example 3**

```
select c11 from t1
where c12 =
    (select c21 from t2 where c22 =
        (select c31 from t3 where c32 = t1.c11))


( nested
     ( t_scan t1 )
     ( subq 1
          ( nested
            ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
               ( subq 2
                    ( t_scan ( table t3 ( in ( subq
2 ) ) ) ) )
                )
          )
     )
)
```

A level 2 subquery nested into a level 1 subquery nested in the main query.

Usage

- The subq operator has two meanings in an abstract plan expression:
  - Under a nested operator, it describes the attachment of a nested subquery to a table
  - Under an in operator, it describes the nesting of the base tables and views that the subquery contains
- To specify the attachment of a subquery without providing a plan specification, use an empty hint:

```
( nested
    ( t_scan t1)
    ( subq 1
            ()
```

```
    )
)
```

- To provide a description of the abstract plan for a subquery, without specifying its attachment, specify an empty hint as the derived table in the nested operator:

```
( nested
    ()
    ( subq 1
        (t_scan ( table t1 ( in ( subq 1 ) ) ) )
    )
)
```

- When subqueries are flattened to a join, the only reference to the subquery in the abstract plan is the identification of the table specified in the subquery:

```
select *
from t2
where c21 in (select c12 from t1)
( nl_g_join
    ( t_scan t1 )
    ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
```

- When a subquery is materialized, the subquery appears in the store operation, identifying the table to be scanned during the materialization step:

```
select *
from t1
where c11 in (select max(c22) from t2 group by
c21)
( plan
    ( store Worktab1
       ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
    )
    ( nl_g_join
        ( t_scan t1 )
        ( t_scan ( work_t Worktab1 ) )
    )
)
```

See also in, nested, table

# t_scan

Description
Specifies a table scan of a stored table.

Syntax
( t_scan *stored_table* )

Parameters
*stored_table*
is the name of the stored table to be scanned.

Return value
A derived table produced by the scan of the stored table.

Examples

```
select * from t1
```

```
( t_scan t1 )
```

Performs a table scan of *t1*.

Usage
- Instructs the optimizer to perform a table scan on the stored table.
- Specifying t_scan forbids the use of reformatting and the OR strategy.

See also
i_scan, scan, store

# table

Description
Identifies a base table that occurs in a subquery or view or that is assigned a correlation name in the from clause of the query.

Syntax
( table *table_name* [ *qualification* ] )

( table ( *correlation_name table_name*) )

Parameters
*table_name*
is a base table. If the query uses the database name and/or owner name, the abstract plan must also provide them.

*correlation_name*
is the correlation name, if a correlation name is used in the query.

*qualification*
is either in (subq *subquery_id*) or in (view *view_name*).

Examples
**Example 1**

```
select * from t1 table1, t2 table2
where table1.c11 = table2.c21
```

```
( nl_g_join
    ( t_scan ( table ( table1 t1 ) ) )
    ( t_scan ( table ( table2 t2 ) ) )
)
```

Tables t1 and t2 are identified by reference to the correlation names used in the query.

**Example 2**

```
select c11 from t1
where c12 =
        (select c21 from t2 where c22 = t1.c11)


( nested
        ( t_scan t1 )
        ( subq 1
            ( t_scan ( table t2 ( in ( subq 1 ) ) ) )
        )
)
```

Table t2 in the subquery is identified by reference to the subquery.

**Example 3**

```
create view v1
as
select * from t1 where c12 > 100

select t1.c11 from t1, v1
where t1.c12 = v1.c11


( nl_g_join
        ( t_scan t1 )
        ( i_scan 2 ( table t1 ( in ( view v1 ) ) ) )
```

Table t1 in the view is identified by reference to the view.

Usage

- The specified derived tables in the abstract plan are matched against the positionally corresponding tables specified in the query.
- The table operator is used to link table names in an abstract plan to the corresponding table in a SQL query in queries that contain views, subqueries, and correlation names for tables.
- When correlation names are used, all references to the table, including those in the scan properties section, are in the form:

```
( table ( correlation_name table_name) )
```

The table operator is used for all references to the table, including the scan properties for the table under the props operator.

See also in, subq, view

# union

Description Describes the union of the two or more derived tables.

Syntax
```
(union
        derived_table1
        ...
        derived_tableN
)
```

Parameters *derived_table1*...*derived_tableN*
is the derived tables to be united.

Return value A derived table that is the union of the specified operands.

Examples **Example 1**

```
select * from t1
union
select * from t2
union
select * from t3


(union
    (t_scan t1)
    (t_scan t2)
    (t_scan t3)
)
```

Returns the union of the three full table scans.

**Example 2**

```
select 1,2
union
select * from t2
```

```
(union
    ( )
    (tscan t2)
)
```

Since the first side of the union is not an optimizable query, the first union operand is empty.

Usage

- The specified derived tables in the abstract plan are matched against the positionally corresponding tables specified in the query.
- The union operator describes the processing for:
  - union, which removes duplicate values and
  - union all, which preserves duplicate values
- The union operator in an abstract query plan must have the same number of union sides as the SQL query and the order of the operands for the abstract plan must match the order of tables in the query.
- The sort step and worktable required to process union queries are not represented in abstract plans.
- If union queries list nonoptimizable elements, an empty operand is required. A select query that has no from clause is shown in example

See also

i_scan, scan, t_scan

# view

Description

Identifies a view that contains the base table to be scanned.

Syntax

view *view_name*

Parameters

*view_name*
is the name of a view specified in the query. If the query uses the database name and/or owner name, the abstract plan must also provides them.

Examples

```
create view v1 as
select * from t1



select * from v1
```

```
( t_scan ( table t1 ( in ( view v ) ) ) )
```

Identifies the view in which table t1 is used.

Usage

- When a query includes a view, the table must be identified using table (*tablename* ( in *view_name* )).

See also

in, table

# work_t

Description

Describes a stored worktable.

Syntax

```
( work_t [ worktable_name
        | (correlation_name worktable_name) ]
)
```

Parameters

*worktable_name*
is the name of a worktable.

*correlation_name*
is the correlation name specified for a worktable, if any.

Return value

A stored table.

Examples

```
select c12, max(c11) from t1
    group by c12
```

```
( plan
    ( store Worktab1
        ( t_scan t1 )
    )
    ( t_scan ( work_t Worktab1 ) )
)
```

Specifies the two-step process of selecting vector aggregates into a worktable, then selecting the results of the worktable.

Usage

- Matches the stored table against a work table in the query plan.
- The store operator creates a worktable; the work_t operator identifies a stored worktable for later access in the abstract plan.

- During plan capture mode, worktables are identified as *Worktab1*, *Worktab2*, and so on. For manually entered plans, any naming convention can be used.
- If the scan of the worktable is never specified explicitly with a scan operator, the worktable does not have to be named and the work_t operator can be omitted. The following plan uses an empty scan operator "( )" in place of the t_scan and work_t specifications used in example

```
( plan
    ( store
        ( t_scan titles )
        )
    ()
)
```

- Correlation names for worktables are needed only for self-joined materialized views, for example:

```
create view v
as
select distinct c11 from t1

select *
from v v1, v v2
where ...

( plan
    ( store Worktab1
        ( t_scan ( table t1 (in ( view v ) ) ) )
    )
    ( g_join
        ( t_scan (work_t ( v1 Worktab1 ) ) )
        ( t_scan (work_t ( v2 Worktab1 ) ) )
    )
)
```

See also store, view