Database Cracking: Towards Auto-tuning Database Kernels

Stratos Idreos

Database Cracking: Towards Auto-tunning Database Kernels

ACADEMISCH PROEFSCHRIPT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor
promoties ingestelde commissie, in het openbaar
te verdedigen in de Agnietenkapel
op donderdag 24 juni 2010, te 14:00 uur

door
Efstratios Ydraios
geboren te Lesvos, Griekenland

Promotiecommissie

Promotor:	Prof. dr. M.L. Kersten
Copromotor:	Dr. S. Manegold
Overige Leden:	Prof. dr. ir. A.W.M Smeulders
	Prof. dr. P.M.G. Apers
	Prof. dr. H. Garcia-Molina
	Dr. S. Chaudhuri
	Prof. dr. P. Klint

Faculteit der Natuurwetenschapen, Wiskunde en Informatica

The research reported in this thesis has been partially carried out at *CWI*, the Dutch National Research Laboratory for Mathematics and Computer Science, within the theme *Database Architectures and Information Access*, a subdivision of the research cluster *Information Systems*.

The research reported in this thesis has been partially carried out as part of the continuous research and development of the MonetDB open-source database management system.

SIKS Dissertation Series No 2010-29.
The research reported in this thesis has been carried out under the auspices of *SIKS*, the Dutch Research School for Information and Knowledge Systems.

The research reported in this thesis was partially funded by the BRICKS project.

ISBN 978-90-6196556-5

Contents

Chapter 1

Introduction

1.1 The Information Era

Information has always been a powerful thing. Gathering, managing, accessing and analyzing information has evolved to be a critical issue for the success of any kind of commercial or government organization. Telecommunications, banking systems, hospitals, etc., they all depend on data management, requiring various crucial properties, e.g., fast data access times, security and accuracy. Even entire scientific fields crucially depend on how efficiently scientists can interpret data in order to find meaningful patterns among huge piles of data. Nowadays, with the advent of rich media data, even a single person's life represents a challenging data management task with tons of music, photos, video, to store, share and organize. No wonder many argue we are living in the information era; economy, personal, social and scientific evolution, they all crucially depend on data manipulation.

1.2 Database Management Systems

The importance of data lead to the creation of the data management research community which over the last roughly 40 years has achieved significant breakthroughs in multiple aspects of data handling. The early research prototypes quickly became commercial or open source products which we now refer to as Database Management Systems (DBMSs). Nowadays, a DBMS is the heart of any major information system. One way or another, we all interact with

database systems in our daily life, i.e., when we withdraw cash from any ATM around the world, when we shop from any on-line or physical store, etc. In fact, what we are doing in these cases is that we are posing *queries* to the DBMS which is responsible for the relevant data.

The success and wide adoption of DBMSs is mainly due to their *generic* design. A DBMS provides a universal backbone where any kind of data can be stored and accessed. A database user does not have to worry about *how* the data will be stored or how his/her queries will be evaluated. The actual data is described by the user as collections of *tables* where each table contains multiple *tuples* of data. Each tuple in turn contains multiple *attributes* and has a unique *key* that helps us to distinguish between different tuples. For example, an employee table in the database of a company would contain one entry for each one of the company's employees. Each tuple would contain specific information, i.e., a number of *attributes* for a given employee, e.g., name, address, telephone number, etc.

This way of describing data is a *declarative* one and is the same with how us humans actually think of information and how we organize it in our minds. The system will then internally find the proper way to physically store the data. This is in simple words what we refer to as the *relational model* which revolutionized the database field (Codd, 1970). In addition, the DBMS query interface is based on a generic and declarative language, SQL, allowing non expert users, i.e., non programmers, to pose queries and exploit the power of a DBMS by simply declaring *what* they want from the system, i.e., give me all employees that work in the company for more than ten years. The system subsequently is responsible to apply the proper algorithms and procedures for this specific query, for this specific data set, given the current system status, etc. This way, non programmers can efficiently use a database system while at the same time a single DBMS can be used for multiple different application scenarios significantly simplifying both the system development and the adoption of the technology.

Another crucial factor in the rapid and successful evolution of DBMSs is their standardized internal architecture, i.e., every system contains the same abstract high level components. These components have more or less the same motivation, the same goal, and they interact in similar ways across all systems. This is of crucial importance as over the years it allowed researchers to talk the "same language", i.e., refer to the same problems, port techniques from one system to another and quickly transfer research ideas into commercial products.

The major components of a DBMS include, (a) the parser, (b) the query optimizer and (c) the execution and storage engine. The parser is responsible for

syntactically analyzing incoming queries. The query optimizer is then responsible for finding the most appropriate *query plan*, i.e., the proper way to execute a query and finally, the execution engine is responsible for actually evaluating a query over the properly stored and maintained data.

Database systems are very complex pieces of software and database research spans over a large number of subareas, each one focusing on specific aspects of a DBMS. Characteristic examples include transaction management, security, distributed processing, etc. Most notably query optimization and physical design have attracted a tremendous amount of attention from the research community and are more relevant with the subject of this thesis. Below we discuss these topics in more detail motivating the research path undertaken here.

1.3 Query Optimization

As we discussed earlier, one of the key points that lead to the wide adoption of database systems is the declarative nature of the interface between users and the system, i.e., the user says what she/he wants and the system automatically finds the best possible way to satisfy the user's request.

The query optimizer is the central point here. This is the component of a DBMSs that on-the-fly analyzes multiple different ways of how a query can be processed and picks the best possible candidate given the existing knowledge. It analyzes the best possible order in which the various actions should be performed and it also finds out what is the best algorithm to actually perform each one of these actions.

In database terminology such actions/steps are called *operators* and a query plan is simply a sequence of operator calls. For example, the select operator selects all tuples from a given table that satisfy a predicate. Typically, a DBMS contains a large collection of operators, while each operator may be implemented in various different ways using different algorithms and exploiting different kinds of data structures. Each one of these implementations is suitable for a different situation and the query optimizer is responsible for calling the proper operator in the proper plan not only to get the correct result but crucially to get the best possible performance as well. When we refer to performance in a database system, we typically refer to *how fast* a system can evaluate a query and create the respective answer.

1.4 Physical Design: Something is not Right!

Over the years, the database community has come up with very sophisticated techniques to improve query optimization. However, no matter how smart a query optimizer becomes and no matter how efficient our individual operator implementations and algorithms become, there is always the inherit limitation of the underlying physical design, i.e., *the way you can access the data depends on how the data is actually physically organized.*

This is exactly what we refer to as physical design. Modern database systems allow the creation of *indices* that can be exploited to improve performance. An index is a copy of *part* of the base data of a given table. Its property is that it stores data in an *order* that is useful for a given class of queries, i.e., it allows for more efficient access patterns and it also typically allows the system to immediately focus on only part of the data by quickly finding out that the rest of the data is not relevant for the given query. This way, queries can be evaluated efficiently by reading the data from the respective index instead of reading the actual table. For example, a query that requests all names of employees which are younger than 40 years old can be evaluated using an index where the employee table is sorted on the employee age. Otherwise, the system has to analyze the employee table tuple by tuple, checking the age of each individual employee.

This way, having the proper indices at hand when the relevant queries arrive can give a huge performance boost. However, creating indices is not a straightforward task. Which indices to create, when to create them and when to use them are decisions that have to take into account multiple different and complex parameters that often continuously change as the incoming queries and stored data evolve. For example, indices bring an extra storage overhead and thus an unbounded index creation strategy to satisfy every single possible request will quickly lead into storage issues. Moreover, indices have to be kept up to date with updates in the base data, i.e., the actual data on the original tables. Every time the data changes, this change has to be reflected into the indices as well, otherwise using these indices will lead to wrong results. The more updates arrive and the more indices we have, the higher the maintenance overhead.

Thus, the task of finding the proper physical design, i.e., the proper collection of indices is a complex problem with numerous tradeoffs to consider. For this reason, it is a task that in modern DBMSs it is assigned to database experts, called database administrators (DBAs). *Here is where the chain of declarative properties breaks!*

Users still say *what* they want from the system in a declarative way, but in

order to achieve the ultimate performance, a DBA will behind the scenes by hand optimize the physical design given the specific user's needs, i.e., given the data and queries used. We typically refer to the collection of data and queries as the *workload*. The task of the DBA is to "understand" the workload and "adapt" the physical design accordingly, i.e., decide when it is a good time to create which indices. In addition, given that the workload might change over time, the DBA is called to continuously *monitor* the system performance and if necessary decide to change the physical design again in the future.

More recently, advances in the database field have lead to the design of *auto-tuning tools*. Such a tool is an external component whose goal is to significantly simplify the task of the DBA, i.e., such tools can off-line or even on-line monitor a system's performance given a workload and provide the DBA with statistics and good recommendations of how the physical design should look like if the workload follows the current patterns. These are valuable tools that allow the DBA to worry less about the mechanics of monitoring and more about taking the proper decisions by carefully putting all available knowledge together. Even though these tools represent a huge step forward in simplifying modern systems, the inherit problem of slow reactions to (random and frequent) workload changes and the bottleneck of keeping humans in the loop are still there.

1.5 Self-organization

The above model worked fine for several years, but as the requirements of new applications and scenarios evolve it is evident that it does not scale. The recent trends in information technologies show that the state of the art database techniques are inadequate to handle the new challenges; large amounts of continuously renewed data and complex access patterns (not known in advance) cannot be efficiently processed by the current *static* systems. The task of DBAs has become a horrendous one, with tons of ever changing information and parameters to consider especially as the systems themselves become more and more complex with more and more knobs to tune. This can be seen in the ever expanding web services where millions of users request continuous concurrent access in a continuously expanding information base. Most importantly scientific databases, e.g., astronomy, biology etc. typically expand their base data on a daily basis leading to huge data sets while scientists look for arbitrary patterns to understand the data. These are characteristic examples of scenarios where both the enormous amount of base data and the unpredictable query load behavior, i.e., focusing on different parts of the data over time, using continuously

changing patterns, etc. makes it harder and harder to meet the challenges.

Typically a brute force approach is used, i.e., more and more hardware support is thrown in to increase the processing power and parallelize the query processing actions. However, these are limited, temporary and expensive solutions. Moreover, the importance of physical design always remains crucial and it is typically the critical factor that defines performance. This way, it is still in the hands of DBAs to deliver and maintain a system with the desired performance. Smart, automatic, adaptive and content-aware physical design and data access is the key to move forward.

A database system should just be given the data and queries in a declarative way and the system should internally take care of finding not only the proper algorithms and query plans but also the proper physical design to match the workload and application needs. This is exactly the vision of *self-organization* research and the topic of this thesis, i.e., to design systems that automatically adapt to the access patterns by *selectively* and *adaptively* optimizing the data set purely for the workload at hand. Ideally, this direction will significantly reduce or even remove the role of DBAs leading to systems that can completely automatically self-tune. Performance is not the only critical parameter here; *simplification* of system administration is the key towards truly scalable systems to meet the modern challenges.

1.6 DB Cracking: Towards DBA-free Systems

In this thesis, we explore a radically new architecture, called *database cracking*. It represents a radical departure from what is considered textbook standard in the area, bringing a lot of challenges to resolve but also huge opportunities to harvest.

The main innovation is that the physical data store is continuously changing with each incoming query q, using q as an advise on how data should be stored.

The ultimate goal of database cracking is to build the first truly self-organizing database system that will continuously and automatically adapt to workload changes. Cracking completely removes the need for human administration. Most notably cracking is not an auto-tuning tool, i.e., it is not an external piece of software to help with system administration. Instead cracking represents a new internal kernel design by introducing new ways of storing and accessing data. This way, the very way data is stored and subsequently accessed by

queries is continuously changing to adapt to the workload and to converge to the ultimate performance.

Database cracking is being studied, designed and developed as part of the MonetDB system. MonetDB is an open-source column-store DBMS with multiple innovations in its core design. Database cracking is disseminated as part of the MonetDB product family available via `http://monetdb.cwi.nl/`.

1.6.1 The Basics

Database cracking is a fully on-line approach aimed towards dynamic environments with rapid workload changes, updates and no or little idle system time to devote to preparation and monitoring steps. It sets a new query processing and adaptation paradigm.

Cracking continuously changes the way data is stored by bringing often used data physically closer to improve access patterns.

Let us give a simplified example using a simple selection query. Cracking is applied at the attribute level, thus a query results in physically reorganizing the column (or columns) referenced, and not the complete table. Assume a query that requests $A < 10$ from a table. A cracking DBMS clusters all tuples of A with $A < 10$ at the beginning of the column, pushing all tuples with $A \geq 10$ to the end. A future query requesting $A > v_1$, where $v_1 \geq 10$, has to search only the last part of the column where values $A \geq 10$ exist. Similarly, a future query that requests $A < v_2$, where $v_2 < 10$, has to search only the first part of the column.

The actual design is much more complex of course especially for more complex queries and updates. We will discuss these in more detail in the following chapters. The main idea is that each query is interpreted not only as a request for a particular result set, but also as an *advice* to crack the physical database store into smaller pieces and organize the data in a different way reflecting the current workload.

The terminology "cracking" reflects the fact that the database is partitioned (cracked) into smaller and manageable pieces. In fact, cracking happens at the operator level. Every crack operator, i.e., selection, tuple reconstruction, join, etc. internally results in physical reorganization actions that can be exploited by any future crack operator in the current or future queries.

1.6.2 Thinking Outside the Box

Cracking naturally provides a promising basis to attack the challenges described in the beginning of this chapter. The really revolutionary part is the fact that we actually physically reorganize the base data *while* processing the queries. Traditionally, query processing time is considered a "holy" time, i.e., do minimum actions to have the best possible reaction time. Cracking shows that investing in some lightweight reorganization, based on what the queries need, can have both short term and long term benefits but most importantly it creates a self-organizing behavior. Without the need for monitoring and preparation steps, each query processed adds a small part in improving the data organization, collectively creating a physical design that matches the workload at the time this workload is actually active. No external (human) administration or a priori workload knowledge is required and no initial investment is needed to create index structures.

1.6.3 Contributions

The contributions of this thesis can be summarized as follows:

(1) **Database Cracking.** We introduce database cracking, a new query processing paradigm towards truly self-tuned systems. Cracking requires zero human input, no a priori workload knowledge and no idle time to prepare.

(2) **Updates in a Cracked Column-store.** We show that cracking can maintain its properties even under frequent and high volume updates by absorbing updates in a self-organizing way.

(3) **Self-organizing Tuple Reconstruction.** We show how to overcome the tuple-reconstruction issues caused by the continuous physical reorganization and at the same time provide the optimal tuple-reconstruction performance for a column-store achieving similar behavior to presorting but without the restriction that come with sorting.

(4) **Crack Joins.** We show how to exploit and even enhance the cracking knowledge gained during past queries for efficient join processing in future ones via a new class of joins algorithms, the crack joins.

(5) **Adaptive Indexing Hybrids.** We study a new technique, termed adaptive merging that was inspired by database cracking and we design the first hybrid indexing approach that blends ideas from both adaptive merging

and cracking leading to very flexible indexing directions and a whole new family of adaptive indexing alternatives.

(6) **The Big Picture.** Finally, we carve the complete research space opened by database cracking towards the vision of a fully functional DBA-free self-organizing DBMS that can automatically adjust to arbitrary workloads.

1.6.4 Published Papers

The material in this thesis has been the basis for a number of publications in major international refereed database venues.

(1) Stratos Idreos, Martin Kersten and Stefan Manegold. Database Cracking. In Proceedings of the 3rd International Conference on Innovative Data Systems Research (**CIDR**), pages 68-78, Asilomar, California USA, January 2007

(2) Stratos Idreos, Martin Kersten and Stefan Manegold. Updating a Cracked Database. In Proceedings of the 27th ACM **SIGMOD** International Conference on Management of Data, pages 413-424, Beijing, China, June 2007

(3) Stratos Idreos, Martin Kersten and Stefan Manegold. Self-organizing Tuple Reconstruction In Column-stores. In Proceedings of the 29th ACM **SIGMOD** International Conference on Management of Data, pages 297-308, Providence, Rhode Island, USA, June 2009

(4) Stratos Idreos, Martin Kersten and Stefan Manegold. Adaptive Joins for Adaptive Kernels. Submitted for publication at the moment of printing this thesis.

(5) Stratos Idreos, Stefan Manegold, Goetz Graefe and Harumi Kuno. Adaptive Indexing. Submitted for publication at the moment of printing this thesis.

1.7 Thesis Outline (How to read)

The rest of this thesis is organized as follows. First, Chapter 2 discusses research work that is relevant to cracking. It points out to how database cracking was inspired by past experiences, how it differs from past approaches in various novel ways and how it can potentially itself inspire further developments in a number

of research areas. In addition, Chapter 2 provides a detailed discussion of our development and experimentation platform, MonetDB.

Then, Chapter 3 introduces the basic cracking architecture and provides all motivation and thinking that lead to the initial cracking work. It reflects the fundamental adoption of cracking in a complete system providing the basic cracking algorithms, demonstrating the initial performance and the initial cracking operators. It shows that on-the-fly physical reorganization is possible leading to a continuously improving performance by needing to analyze less and less data as more queries arrive.

After reading Chapter 3, the rest of the chapters can be read more or less independently. All techniques work in harmony together but regarding reading and understanding there are no real dependencies among them. The only hard prerequisite for each chapter is a good understanding of the basics in Chapter 3 while indeed Chapter 8, that unveils the future research paths opened by database cracking, is best understood once all material is read.

Chapter 4 shows how to maintain the cracking knowledge during updates. Updates is a crucial issue for any kind of improved physical design as transferring the updates to potential indices or other data structures is an added overhead. For cracking it represents a big challenge due to its continuous physical reorganization nature. Numerous new algorithms and techniques are shown that allow us to successfully maintain cracking columns by adaptively and selectively updating only what is really necessary and only when it is really necessary. The cracking knowledge is kept alive at a minimum cost allowing the system to provide the improved cracking performance even after and during updates.

Then, Chapter 5 introduces a more advanced cracking architecture by effectively pushing cracking deeper into the kernel design. Multiple partial cracker maps are continuously, on demand and on-the-fly kept aligned to ensure that tuple reconstruction, the most important operator in a column-store, is always performed using sequential access patterns. Expensive tuple reconstruction via joins is replaced with self-organizing tuple reconstruction via continuously cracking the various columns. Cracking achieves for tuple reconstruction a similar performance to a presorted column-store but without the requirements for a priori workload knowledge, idle time to pre-sort and the need for infrequent updates.

Chapter 6 demonstrates how the cracking knowledge gained during selection and tuple reconstruction can be successfully exploited for efficient join processing. A new class of join algorithms not only exploit past knowledge gained by past queries on the same columns, but also via on-the-fly cracking during join processing, they introduce even more cracking knowledge and thus boosting the

self-organizing properties of the system.

Then, Chapter 7 discusses adaptive merging, a new technique inspired by database cracking and invented by Goetz Graefe and Harumi Kuno. We provide a column-store design and implementation of adaptive merging and an evaluation against database cracking pointing on the design choices and the benefits and pitfalls of both techniques with respect to the workload properties. Then, we design a hybrid technique that provides the first step in blending the positives from both adaptive merging and cracking, while at the same time it opens a plethora of future work for hybrid designs that can adapt to a larger variation of workload patterns.

Chapter 8 concludes the thesis and discusses some of the most prominent future research goals for database cracking. It sets a complete research path opened by cracking, demonstrating that the vision of a truly crack-based system is a not only a realistic one but it also represents a full of potential research direction that applies to and affects every corner of database research. Topics include concurrency control, optimization, external cracking, adaptive denormalization opportunities, distributed and multi-core cracking, etc.

Chapter 2

Related Work and Background

It is clear that, by aiming at the design of a completely self-organizing DBMS, cracking touches upon every angle of database design. As such, and even though cracking introduces a completely novel query processing paradigm, the bigger picture of the cracking initiated research is related to numerous important research areas within the database field. If one looks carefully at the technical details, cracking has been heavily inspired by past research while at the same time it introduces numerous innovations and when all put together, the grand vision of a self-organizing DBA-free DBMS becomes a reality.

In this chapter, we briefly discuss some of this related work and its connection with cracking. We discuss about column-oriented versus row-oriented processing and how cracking found a more fertile initial environment in column-stores due to the simpler data layout as well as how cracking improves certain core column-store processing issues, e.g., tuple reconstruction and updates. We also relate cracking to auto-tuning tools and the use of materialized views and indices, motivating its design goals towards the new challenge of completely auto-tuned systems. In addition, we discuss how cracking can be exploited for join processing bringing a new angle to the classic partitioned join problem. We then motivate cracking for distributed environments where its self-tuned logic can be exploited for better data placement.

Finally, we introduce the MonetDB database system, which is our implementation and experimentation platform.

2.1 Row-oriented Storage and Data Access

As it should be already clear by our previous discussion, the way data is stored in a DBMS defines/restricts the way we can process queries. All together, storage and access methods define the possible optimizations we can think of and implement over an existing architecture.

Traditional database systems *store and process* data *one tuple* at a time, i.e., one row of a table at a time. This way, we will refer to them with the general term, *row-stores*. In subsequent sections, we will discuss in detail about *column-stores* as well which form our experimentation environment for cracking.

The storage layout in a row-store is typically based on *pages*. Each page holds a certain amount of data, e.g., 8K, and contains a number of rows from a given table. Given the variety of possible data types in a table, we need to maintain a number of metadata entries for each page, for each row in a page and for each attribute in a row. For example, we need to know information regarding data sizes, starting position of attributes and rows, etc. All this is necessary so that we can "navigate" through a page.

The processing model in a row-store is typically based on the volcano ideas, i.e., the query plan is given one tuple at a time. Each tuple goes all the way through every operator in the plan, before we move on to the next tuple. For example, assume the following simple query.

$$\mathsf{select\ sum(R.b)\ from\ R\ where\ 5} \leq \mathsf{R.a}$$

A row-store plan would go one by one through all tuples of table R. For each tuple it would first call the select operator to evaluate the predicate of the where clause. In the case that the given tuple satisfies the predicate, it would then call the sum operator.

This way of query processing adds to the need for this rather complex data layout in row-store pages. During query processing, we continuously need to move on to the next attribute of a row which means we need to know how many bytes we need to read from the row and from which point exactly. We also need to call the next operator in the plan and be aware of the data type which can of course be different than that of the previous attribute. Then, once we are done with one row, this has to happen again with the next row and so on.

2.2 Column-stores

Row-store technology has been the basis for all major commercial products. It prevailed in the early years for good reasons, e.g., organizing data in the form of tuples allows to easily load, update and process all relevant data given a database entry. This kind of processing was more typical when databases where mostly used for online transaction processing (OLTP).

Over the years, the application needs changed. In addition to OLTP, applications now critically need the ability to handle analytical queries for online analytical processing (OLAP). This kind of queries do not always need to process full tuples. Instead, they focus on analyzing a subset of a table's attributes, e.g., running various aggregations to understand and analyze the data. For this kind of applications a column-store architecture seems more natural which lead to the design of a number of very interesting systems, e.g., MonetDB (MonetDB, 2009), MonetDB/X100 (Boncz et al., 2005) and C-Store (Stonebraker et al., 2005).

These systems are inspired by the Decomposition Storage model (DSM) (Copeland and Khoshafian, 1985). Column-oriented DBMSs store data one column at a time as opposed to one tuple at a time. This brings the obvious benefit of allowing a system to benefit a lot in terms of I/O for queries that require only part of a table's attributes. For example, assume a table representing employees in a company's database. This table will typically consist of numerous attributes, i.e., first name, last name, address, telephone number, employee ID, salary, hire date, department, etc. Now imagine a series of queries with a goal of analyzing the data, e.g., give me all employees that work in the company for at least 5 years, give me all employees with a salary more than X, give me the average salary, give me the average working experience, etc. This kind of queries need to see only part of the whole table. However, in a row-store the default action (assuming no indices are present) would still be to load the whole table from disk to memory. A column-store, on the other hand, needs to load only what is relevant to the query. However, a column-store is much more than simply storing data one column at a time!

Another strong point of column-stores is the increased opportunities for compression (Abadi et al., 2006; Zukowski et al., 2006). Physically storing together columns, brings similar data closer, i.e., data of the same type with high chances of having the same or similar values. This way, significant compression levels can be achieved. For example, dictionary compression in a row-store would typically happen at the page level, i.e., there would be one dictionary for each page and it would encode every possible value in this page. However, by containing

full tuples with multiple different data types and unrelated values, the chances for compression are restricted. In a column-store though, a page contains only values of a single attribute, increasing the chances of finding good compression cases. Similar arguments stand for run length encoding as well.

On the other hand, the obvious drawback of a column-store setting, is the on-the-fly *tuple reconstruction* needed to bring the necessary columns back in a tuple format. Each tuple reconstruction is a join between two columns based on tuple IDs/positions and becomes a significant cost component in column-stores especially for multi-attribute queries (Idreos et al., 2009; Abadi et al., 2007; Harizopoulos et al., 2006; Manegold et al., 2004). Whenever possible, position-based join-matching and sequential data access are exploited. For each relation R_i in a query plan q, a column-store needs to perform at least $N_i - 1$ tuple reconstruction operations for R_i within q, given that N_i attributes of R_i participate in q. This is obviously a tradeoff depending on various parameters, i.e., the number of attributes from a table R that are relevant to a given query, how wide R is, etc. In fact, the definition of tuple reconstruction is not that straightforward as it actually depends on the way a column-store is implemented.

2.2.1 Early Tuple Reconstruction

The simplest way to implement a column-store would be to extend an existing row-oriented system, i.e., create a schema with multiple single column tables. The drawback here is that we are still maintaining the complex page format requiring all kind of metadata maintenance. The next step is to have a dedicated column-store implementation but still maintain the typical N-ary processing model, i.e., store data one column at a time with a simplified storage layout but process data one tuple at a time in the same way as row-stores do. Tuple reconstruction here is the act of gluing tuples together immediately after loading data from disk, i.e., we load only the column necessary for a given query, we then immediately physically glue the columns in a tuple format and then process them as in a row-store setting. We call this *early* tuple reconstruction. The benefit is that each tuple is reconstructed only once per column and that the positional alignment of the base columns is exploited. Then, potentially a large number of not required data is carried on through the query plan.

2.2.2 Late Tuple Reconstruction

As we already mentioned, a column-store is much more than simply storing data one column at a time. In fact, some of the most significant benefits,

optimizations and research questions come from *processing data one column at a time*. This is actually how all advanced column-stores are designed (Stonebraker et al., 2005; Boncz et al., 2005; MonetDB, 2009) while (Abadi, 2008) provides a nice analysis on how a pure column-store design is much more beneficial than the previously mentioned approaches.

This approach allows the query engine to exploit CPU- and cache-optimized *vector-like* operator implementations throughout the whole query evaluation allowing to minimize function calls, type casting, various metadata handling overheads, etc.. All operators are heavily optimized by operating directly on columns instead of tuples. Assume a query q that first selects a range over attribute A_1 of R. For example, the select operator is loaded only once and operates directly on the A column in one go. Then, for *each* subsequent operator (select, join, project etc.) on a different R attribute, A_2, in the query plan of q, the system first performs *logical* tuple reconstruction for A_2, i.e., get in a column form the proper/qualifying A_2 values and in the correct order. All operators benefit significantly by operating on dense data (of similar type) in a vector-like model. In addition, the CPU caches are filled only with relevant data allowing to exploit modern CPUs to the maximum. Furthermore, only the required data is carried through the query plan.

Tuple reconstruction here is called *late* tuple reconstruction and allows to exploit the column-store architecture to the maximum. During query processing, tuple "reconstruction" merely refers to getting the attribute values of qualifying tuples from their base columns *as late as possible*, i.e., only once an attribute is required in the query plan. Intermediate results are also in a column format and are glued together in a tuple N-ary format only prior to producing the final result. In fact, actual physical reconstruction is not even necessary. As far as the user is concerned, viewing the results in a tuple format is mainly a presentation issue, i.e., the system merely needs to print/output the result columns in a tuple format.

The overhead is that a column/attribute might need to be reconstructed more than once. Also, in some cases (unless a preparatory step is taken), exploiting sequential access patterns during tuple reconstruction is not possible since many operators (joins, group by, order by etc.) are not *tuple order-preserving*. For example, joins, group by and order by operators destroy the tuple order in intermediate results resulting in a significant performance decrease for tuple reconstruction.

2.2.3 Updates

Updates have been coined as one of the major differences between column- and row-oriented systems. Updating a column-store seems naturally as a heavier operation since we need to touch multiple data areas to update a single tuple whereas in a row-store this can be done in a single step. Even further, a column-store architecture that supports "projections", i.e., multiple data copies sorted in different orders, suffers even more from updates as it needs to update even more data as well as maintaining the order in each projection.

2.2.4 Cracking on Column-stores

Cracking was inspired in a column-store environment as it inherently gives maximum flexibility by allowing to manipulate and reorganize columns independently. Storing columns separately means that we can independently reorganize each column without affecting the rest. Furthermore, the bulk processing model allows to completely operate on columns in one go minimizing the overhead of on-the-fly reorganization since we are "seeing" all to be reorganized data anyway.

As we will show in subsequent chapters cracking even managed to significantly improve not only the selections but also the late tuple reconstruction performance of column-stores even for dynamic and frequently updated environments.

2.2.5 Future Column-store Research

Column-stores are a relatively young research field compared to how much research has gone into row-store specific issues. In this sense, one should only expect that there is a vast amount of research opportunities here in many different areas.

Regarding the core architectural design and tradeoff, i.e., store one column at a time or store one tuple at a time, this is an area that future research is called to study. Definitely a pure column-store design seems more flexible and carries a significant amount of benefits. However, in the same way a pure row-store design is the one extreme of the problem, a pure column-store is the other extreme.

In research labs we have seen a number of very interesting hybrid directions, i.e., (Ailamaki et al., 2001; Hankins and Patel, 2003; Ramamurthy et al., 2002; Zhou and Ross, 2003; Shao et al., 2004). In these works, data placement is a

compromise between rows and columns trying to create systems that span over a broader application area. Studying how cracking could be applied in such data layouts, the side-effects and the potential is a very interesting topic of future work.

In addition, a number of interesting alternatives have emerged as part of the MonetDB research and continuous development focusing more on the execution part. One is the MonetDB/X100 system (Boncz et al., 2005) where the main idea is that columns are split into multiple physical pieces that comfortably fit in memory. This allows for a hybrid processing model, i.e., combining bulk processing with volcano processing. MonetDB/X100 consumes pieces of full columns at a time instead of the complete columns in one go. Partial cracking (Idreos et al., 2009), i.e., physically splitting columns into multiple pieces and cracking each piece separately, is introduced later in this thesis and provides the basis for cracking to be exploited in these environments too. Another idea goes as far as dynamically gluing columns in a tuple format if during processing it is detected that a collection of operators can actually be processed more efficiently in an N-ary way (Zukowski et al., 2008). Dynamically splitting and gluing columns continuously through the query plan depending on query and data is envisioned as a very promising research path.

2.3 C-Store

The recently proposed system C-Store (Stonebraker et al., 2005) is related to cracking in many interesting ways. First, C-Store itself is also a brave departure from the usual paths in database architecture. C-Store is a column oriented database system and its main architecture novelty is that each column/attribute is sorted and this order is propagated to the rest of the columns. Multiple projections of the same relation can be maintained, up to one for each attribute. The sorted order is exploited for fast selections while the alignment across the columns of each projection is exploited for fast tuple reconstruction. To handle the extra storage space required by the multiple projections compression is extensively used (Abadi et al., 2006).

2.3.1 Cracking vs C-store

Thus, C-Store also physically reorganizes the data store. In many ways, a comparison between a cracking approach with the C-Store one would contain the arguments used in the comparison with the sort and an index based strategy.

The main characteristic of cracking is its self-organization based on the query workload, doing just enough, touching enough of the data each time, and being able to change the focus to different parts of the data dynamically. These properties are not explored in the C-Store approach. C-Store needs to prepare for the workload, by creating and maintaining the proper projections. Another crucial factor is that of updates. As we show in this thesis, updates can gracefully be handled in a cracking setting. On the other hand, C-Store targets read-only or infrequently updated scenarios as the cost of on-the-fly maintaining multiple fully sorted copies of data is prohibitive.

A very interesting aspect of C-store is how compression is exploited especially for the column in each projection that is fully sorted. In this case, the system can have the best possible run length encoding compression. Compression is an open topic for cracking. We can obviously have all the benefits of column-store related flexibility by compressing one column at a time. Where it becomes very interesting is when physical reorganization decisions within the crack operators take compression into account when they decide which values to physically move closer (see Chapter 8 in our discussion for the grand picture of cracking research).

2.4 Indices

Other than sophisticated query plan optimizations, the predominant way of improving performance in row-stores is the use of *indices*. An index contains all or part of a base table's attributes and its main purpose is that it stores data in a different way than the base table, i.e., in way that is actually preferable for a given set of queries. When an index contains all table attributes is called *clustered*. We can typically have at most one clustered index per table which actually reflects a reorganization of the table itself and does not require any extra storage. Then depending on the design we can have a large number of *non-clustered* indices where we actually make partial copies of the base data using only part of the attributes every time.

The index *key*, i.e., one or more of the table's attributes, defines the order the index data is stored. We can have multi-attribute keys defining a primary order, a secondary order, etc.

By using indices, queries can be processed very efficiently. Other than utilizing a useful order of the data, an index also helps to improve with the loading of data. For example, an index typically contains only a small part of the attributes of a wide table, i.e., a query using this index avoids loading the rest of the data.

The typical index used in database systems is the B-tree (Bayer and McCreight, 1972; Comer, 1979), i.e., the pages are logically organized in a tree form where the index key defines how the various pages are linked. Typically, the actual physical storage contains a number of tree pages that during query time help us to navigate through the tree based on a key. Then, the main storage is the actual data pages that reflect the leaf pages of the tree and contain the data in the proper order.

2.4.1 Cracking vs Indices

One of the main questions is how cracking compares to traditional indices. Another interesting question is how cracking compares against a full sort strategy, i.e., fully sort a column instead of continuously and incrementally cracking it.

Assume an environment where it is known *up-front which data is interesting* for the users/queries, i.e., which single (combination of) attribute(s) is primarily requested, and hence should determine the physical order of tuples. Assume also that there is the luxury of *time and resources* to create this physical order *before* any query arrives and that there are no updates or the time difference between any update and an incoming query is sufficient for maintaining this physical order (or maintaining the appropriate indices that provide an order, e.g., B-trees). If all these are true, then sorting, or any kind of appropriate index is a superior strategy.

Cracking is not challenging any strategy in such conditions. Instead, as we already mentioned in our motivation in Chapter 1, cracking targets the vision of self-organizing databases to autonomously adapt in dynamic environments where:

- there is not any knowledge about which part of the data is interesting, i.e., which attributes and ranges (and with what selectivities) will be requested.
- the workload is changing often and/or with an unpredictable pattern.
- there is not enough time to prepare the complete physical design.
- there is not enough time to restore or maintain the physical order after an update.

We will clearly show that cracking is a lightweight operation and that it needs no up-front knowledge or idle time to achieve fast data access.

2.5 Auto-tuning Tools

The first steps towards the vision of self-organization have already been made. In the past years, there has been a lot of effort in designing tools to prepare the database for the anticipated workload, e.g., (Agrawal et al., 2004; Bruno and Chaudhuri, 2006; Chaudhuri and Narasayya, 1997; Qin et al., 2007; Schnaitter et al., 2006; Valentin et al., 2000; Papadomanolakis et al., 2007; Zilio et al., 2004). These tools fall under the umbrella of *auto-tuning* advisors/tools with paper (Chaudhuri and Narasayya, 1997) being the pioneering work to bootstrap this area.

In general, this line of work is based on the notion of monitoring the current workload while the system is working or most typically the tools are offline given a representative workload to analyze. After "enough" statistics have been collected, clever algorithms are applied to *predict* the future workload taking into account the past behavior. Then, the system makes a decision on whether the current physical design should change or not. If yes, then a proper physical design is chosen, i.e., indices that should be applied to speed up future queries. This prediction is transferred to the database administrator that makes the final judgment.

2.5.1 What-if Analysis

The core idea is based on a *what if analysis* (Chaudhuri and Narasayya, 1997). The tool is given a set of queries and data to analyze. During the analysis, the tool will observe each query independently and continuously "wonder"; *what if index X was available?* It will then simulate the behavior of the whole query workload assuming index X is available. Candidate indices are considered given the structure of the query. Trying multiple possible indices per query and multiple different index combinations, the tool will try to make the best possible choice for each query but also make choices that have the bigger positive impact for the whole workload. Obviously, auto-tuning tools do not actually create and try out all these indices. Instead, they rely on sophisticated techniques to *predict* the behavior of the system *assuming* an index was available. This way these temporary and not materialized indices are called *what if indices*. Here the query optimizer of the DBMS is called to give a proper cost estimation given a query and a what if index.

Compression can also be considered by auto-tuning tools to boost performance. In (Idreos et al., 2010a), we discuss the problem of *what if compression analysis* to study how such tools can leverage compressed indices, i.e., how do

we decide when a compressed index is useful for the workload. Numerous challenging research and technical problems arise. For example, estimating the size and the expected performance of a hypothetical compressed index without actually creating the index itself or touching all its potential contents. And the problem becomes even more complicated once we consider update queries and storage restriction as well.

In general what-if analysis is a very hard problem, and even this off-line approach studied so far, and implemented by major vendors, is a huge step forward requiring to solve hard research and technical problems. The contribution of this line of work is critical, i.e., a DBA can now analyze more in less time leading to better and faster decisions. They simplify and shrink the role of the DBA allowing less room for human errors and allowing the systems to scale better.

However, in practice auto-tuning tools do not make any fundamental changes to the system. They are external tools that try to simplify the role of the DBA, i.e., they do most of the hard work of monitoring and analyzing the various alternatives allowing the DBA to concentrate on making educated decisions of what should actually change. But the basic model of physical design remains the same relying on *workload monitoring and subsequent index creation.*

2.5.2 Cracking vs Auto-tuning Tools

Such an approach works well for off-line environments with a rather stable workload. In this case the following is true.

- There is enough time and resources to spend on monitoring and altering the physical design.
- The future workload can be safely predicted.

However, the vision of self-organizing databases aims at removing these steps such as to also cope with environments where the workload changes *often* and *rapidly.* In such cases, an on-line approach is required to avoid high latency during the migration step from one physical design to another and to avoid the side-effects of a non-successful prediction or those of a rapid workload change. Even more crucially, with auto-tuning tools the role of the DBA is still an important one as the DBA is the one that makes the final decisions. Ideally, a self-organizing system will need zero human input being able to automatically tune and thus quickly and correctly adapt to changing environments. This is exactly the motivation and contribution of this thesis.

Database cracking can be seen as a natural successor of these ideas. It sets a completely different example, though. Self-organization is not anymore a task of an external tool. Instead, it is an integral part of the database kernel; every operator is redesigned to provide a collectively self-organizing behavior. Thus, cracking is not based on a static decision leading to a collection of indices. Instead, it continuously learns how to better organize the data, with each operator used in any query plan helping to adapt a bit better every time. This way, cracking does not need to perform a prediction and suffer from the effects of possible wrong judgments and delays. It reacts immediately on every query and continuously prepares for future queries by on-line physical reorganization. It removes the need for a DBA, it can handle random/unpredictable workload patterns, as well as adapt to storage restrictions and updates.

2.6 Materialized Views

Other than using indices to speed up access times, modern database systems can also exploit *materialized views* (Gupta and Mumick, 1999). A materialized view is a copy of part of the table and represents a powerful tool to improve performance as similar to an index, a materialized view allows the system to restrict data access for queries that are "covered" by a view.

The contents of a view are typically expressed as a database query and the DBA is responsible to create the proper views to match the workload. Naturally, the more complex the definition of a view the less likely it is to be exploited by future queries but the higher the potential benefits for queries that do use the view. Multiple views can be created over the same table and over the same data. The system then has to on-the-fly decide which views (if any) to exploit when a relevant query arrives.

Recent ideas try to exploit views in a more dynamic way. For example, partial views (Luo, 2007) exploit materialized views to give back incomplete but quick answers so that the users can inspect these early results and possibly decide to terminate the current query, e.g., because the current partial result is already enough or because the results indicate that the query can be formulated in a better way. While the user inspects these partial results, the system continues to fully answer the query using the base data. In any case, the goal here is to make the system more user friendly paying only a small performance price as these partial views are only results from recent queries, i.e., these views are still memory resident and can be accessed in a minimal cost. Similar ideas have been also proposed in the context of semantic query caching (Godfrey and

Gryz, 1997; Godfrey and Gryz, 1999).

Another, recent example of a more dynamic behavior is the dynamic materialized views as introduced in (Zhou et al., 2007). The idea here is that not all tuples of a view are relevant at all times. For example, part of a view might not be used so often by queries meaning that the system pays the overhead of storing and maintaining this part of the view but without receiving the expected benefits. In a dynamic view, we can keep only part of the view materialized while the actual materialized tuples in a single view may even change over time as the workload changes, i.e., based on access frequency.

2.6.1 Cracking vs Views

The crucial difference between cracking and materialized views is that cracking is a direction purely designed for dynamic/unpredictable environments. The physical design of a cracking DBMS can also be described by a set of queries. However, in this case, everything happens completely dynamically with no external control needed. Initial creation, maintenance under updates, adaptation when the workload changes, it all happens automatically *as part of query processing*.

Most importantly cracking reacts at the individual operator level. For example, a select operator will independently reorganize the relevant column in a better way so that any future crack operator can exploit it. Similarly a crack join or tuple reconstruction, etc. will bring relevant data together in the respected column. This way, it all happens automatically at the operator and column level as opposed to the tuple and table level by the DBA or an analysis tool.

One of the main "problems" in a materialized view setting is always the question of how we decide which views to exploit with a given query or set of queries. The answer to a given query may be obtained by exploiting multiple materialized views and by combining base data, leading to a stream of very interesting research on complex decision algorithms that determine when to use what. Simplicity is often the key for efficient, flexible and easy to maintain systems. Especially in dynamic environments the problem becomes much harder due to the unpredictability involved. Cracking sets a much simpler and clear paradigm as there are no such decisions involved. Cracking instantly reacts in every query performing the minimal necessary actions. The data used to answer a query are *always* retrieved from the same place, i.e., the cracker maps which are continuously kept aligned and populated purely based on the current workload needs.

2.7 Partial Indexing

Partial indexing has not received much attention as the prime scheme for organizing navigational access. Partial indices have been proposed to index only those portions of the database that are statistically likely to benefit query processing (Stonebraker, 1989; Seshadri and Swami, 1995). They are auxiliary index structures equipped with range filters to ignore elements that should be referenced from the index. Simulations and partial implementations in Postgres have been undertaken.

Cracking goes a step further. Rather than being told by the DBA to index a specific part of the data, cracking automatically analyzes only parts of the data based on the workload. It learns more and more as more queries are being processed. Similarly to a partial index, it has knowledge for only part of the data but cracking can automatically expand/reduce its knowledge as the workload evolves. Moreover, cracking does not simply index data. Instead, it physically reorganizes the relevant data parts to cluster the data, which improves processing significantly.

2.8 Join Processing

One of our contribution in this thesis, is a new set of algorithms to perform joins (see Chapter 6). The join is the most challenging operation in a database system. Through the many years of database research it has received a lot of attention which has lead to the design of specialized algorithms for a large number of cases that might occur in a typical query processing scenario (Graefe, 1993; Mishra and Eich, 1992).

Contemporary DBMSs implement a large number of variants of join algorithms. The main characteristic of an efficient join algorithm, is that by having a structure over the join attributes, e.g., a sort order, a hash table, etc., we can more easily locate matching tuples. Otherwise a nested loop join has to be performed, i.e., for each tuple from the one input we have to check all tuples of the other input. The inputs are then called *outer* and *inner*, i.e., for each tuple in the outer we try to find hits in the inner. This is extremely slow as it means that we have to scan the inner as many time as the number of tuples in the outer.

This way, a nested loop join is typically preferable only if the inner can comfortably fit in the CPU caches in which case a scan is very fast. Otherwise, systems use algorithms that exploit some sort of structure. For example, a sort

merge join, first sorts both inputs (if they are not already sorted) and then walks over the two inputs to find matching results. Exploiting the sorted order means that the merge phase can be performed very fast. A hash join, first creates a hash table over one of the two inputs and then walks over the other side and for each tuple it prompts the hash table for matching values.

Efficient access patterns during the join operation is the key and the actual cost has to be balanced against the preparation effort needed. A nested loops join needs zero preparation but has a high join cost unless the inner is very small. Similarly, a sort merge join has a very steep preparation to sort both inputs but the joining phase is extremely fast exploiting the best possible access patterns as every tuple is accessed sequentially and needs to be loaded at most once. On the other hand, a hash join has a much smaller preparation cost but its joining phase is slower since access to the hash table can create random access patterns which can impact performance significantly if the hash table does not fit in the CPU caches.

This way, for any given join operation in a query plan, the system *chooses* one algorithm depending on various parameters that can affect performance, e.g., the state of the system, memory restrictions, possible existing or beneficial to built data structures over the data to be joined, the size of the inputs, etc. This choice is made based on a cost model that takes into account all these different parameters. It determines whether creating a data structure and using a specialized join algorithm is expected to be faster than using a naive one, or one that requires less effort in preparing the data.

2.8.1 Crack Joins vs Traditional Joins

The crack join techniques proposed in this thesis follow the cracking philosophy towards a completely self-organizing system. They exploit *but also enhance* the partitioning-like information created during query processing by on-the-fly physically reorganizing the inputs in a cache conscious way. They share common ground with and are inspired by the rich literature on join query processing (Graefe, 1993; Mishra and Eich, 1992). Especially, algorithms on partitioning based joins (Manegold et al., 2002; Shatdahl et al., 1994) are very related to crack joins. Most partitioned join algorithms aim at explicitly partitioning (transient) join inputs only for the purpose and benefit of improving the performance of the very join at hand. In contrary, a crack join exploits in the best possible way an already existing and *evolving* data structure, and through on-line physical reorganization achieves to speed up not only the current join operation but also future joins, selections, updates, etc. The self-organizing

nature of the cracker joins does not exist in any of the past algorithms.

2.9 Distributed and Peer-to-peer DBMSs

The characteristic of database cracking is the continuous physical reorganization of data to match the workload. Even though in this thesis, cracking is studied for a single machine environment, it is evident that these ideas can potentially be explored for a distributed or multi-processor setting as well. The database community has done a lot of work in the area of distributed and parallel databases. A large portion of this work is nicely surveyed in (Kossmann, 2000; DeWitt and Gray, 1992; Yu and Chang, 1984). The crucial parameter is always the question of *where data is placed* so that future queries can have better, faster, easier access. Data is partitioned on multiple disks or sites based on some criteria, e.g., hash-based partitioning, range-based partitioning, etc. Cracking does exactly that; it dynamically decides how data should be stored based on incoming queries and thus an extension towards distributed cracking seems a very interesting potential (see Chapter 8 for more detailed discussion on this).

Mariposa (Stonebraker et al., 1996) is one of the most well known distributed database systems and probably the most ambitious attempt to scale to thousands of nodes. The authors based their work on different assumptions from what had happened up to this point in the area of distributed databases and based the protocols of interaction between the various nodes on economic models. Most importantly they took into account the fact that not all nodes in a network are equal in terms of available resources (CPU, storage, network connection etc.) and the fact that data should be able to move from one node to another easily without requiring heavy operations. All this should happen without global coordination while at the same time the network should adapt its behavior according to current status. Most of these issues still remain open research subjects.

Another well known distributed database system is LH* (Litwin et al., 1996). The authors introduce the notion of the scalable distributed data structure (SDDS) which shares a lot with the underline ideas of current structured overlay networks in the sense that a SDDS is maintained in a distributed way even in the presence of node connections, disconnections or failures without centralized coordination.

Other typical examples include the series of work on exploiting hash based techniques to distribute data and queries. For example, (Mehta and DeWitt,

1997) study the problem of evaluating join queries in multiprocessor environments by parallelizing the join operation mainly through hash-based algorithms.

Dynamic environments Given the hard problems that have to be solved and the strict requirements for consistency and correctness, attempts towards distributed databases typically concentrated on stable environments. DBAs would decide up-front how data should be spread on the network to match the workload. This is very much the same as our discussion earlier in this chapter for the use of indices and cracking can potentially have an impact to more dynamic environments.

P2P DBMSs As the years gone by and both the hardware and the applications evolved, a new era of distributed computing emerged, i.e., *peer-to-peer* computing. The goals and assumptions here are rather different, assuming no centralized point of failure, clients contributing actively to query processing, not all data and nodes are guaranteed to be available at all times, random workloads, etc.

Data and query models in peer-to-peer systems were purposely kept simple to concentrate on more critical distributed processing issues in dynamic environments. This lead to a series of very interesting system prototypes. For example, in (Idreos et al., 2004) we study query processing in large *unstructured* networks using models inspired from Information Retrieval. In (Tryfonopoulos et al., 2005), we show how to support this kind of query processing in a more structured environment, i.e., over Distributed Hash Tables (DHTs) (Stoica et al., 2003). As peer-to-peer computing became more and more popular, the community focused on multiple different kinds of processing. For example, in (Liarou et al., 2006), we study RDF query processing over DHTs. All this line of work is based on the notion of properly assigning parts of the data to parts of the network so that when queries arrive we can efficiently process them by spreading the query processing load quite evenly over the network nodes.

As the application started having more requirements, the idea of P2P DBMSs was coined (Bernstein et al., 2002; Gribble et al., 2001), i.e., use relational data and SQL queries in a P2P environment. Then, depending on the application scenario we can possibly relax some of the strict requirements of relational query processing, i.e., it is not necessary to always spend huge amounts of network bandwidth to create a complete answer. PIER (Huebsch et al., 2003) was one of the first attempts to support the full power of SQL over DHTs while in (Idreos et al., 2006; Idreos et al., 2008) we introduced a series of algorithms to perform

incremental continuous join query processing in a DHT environment.

Self-organization in the way data is stored and accessed over the network is an even more crucial property than in a centralized system, i.e., the potential gains in network delays are even more significant than the respective I/O. Cracking is designed for relational query processing and self-organization is its main property. Especially with partial sideways cracking, a cracked P2P DBMS looks as a promising future. With partial cracking columns are adaptively split into multiple physical pieces which the system can process separately, e.g., in separate nodes. In Chapter 8 we have a more detailed discussion on this possibility as well.

2.10 The MonetDB System

In this section, we will briefly describe the MonetDB system to introduce the necessary background for the rest of our presentation. MonetDB is an open-source column-store developed in the database group of CWI in Amsterdam over the past decade.

MonetDB differs from the mainstream systems in its reliance on a decomposed storage scheme, a simple (closed) binary relational algebra, and hooks to easily extend the relational engine. In MonetDB, every n-ary relational table is represented by a group of binary relations, called *BAT*s (Boncz et al., 1998). A BAT represents a mapping from an `oid`-key to a single attribute `attr`. Its tuples are stored physically adjacent to speed up its traversal, i.e., there are no holes in the data structure. The `key`s represent the identity of the original n-ary tuples, linking their attribute values across the BATs that store an n-ary table. For base tables, they form a dense ascending sequence enabling highly efficient positional lookups. Thus, for base BATs, the key column is a virtual non-materialized column. For each relational tuple t of R, all attributes of t are stored in the *same* position in their respective column representations. The position is determined by the insertion order of the tuples. This tuple-order *alignment* across all base columns allows the column-oriented system to perform tuple reconstructions efficiently in the presence of tuple order-preserving operators. Basically, the task boils down to a simple merge-like sequential scan over two BATs, resulting in low data access costs through all levels of modern hierarchical memory systems.

SQL statements are translated by the compiler into a query execution plan composed of a sequence of simple binary relational algebra operations. MonetDB is a late tuple reconstruction column-store; when a query is fired, the

relevant columns are loaded from disk to memory but are glued together in a tuple N-ary format only prior to producing the final result. This way, intermediate results are also in a column format. In MonetDB, each relational operator materializes the result as a temporary BAT or a view over an existing BAT. Intermediates can efficiently be reused (Ivanova et al., 2009). For example, assume the following query:

$$\text{select R.c from R where } 5 \leq \text{R.a} \leq 10 \text{ and } 9 \leq \text{R.b} \leq 20$$

This query is translated into the following (partial) plan:

```
Ra1 := algebra.select(Ra, 5, 10);
Rb1 := algebra.select(Rb, 9, 20);
Ra2 := algebra.KEYintersect(Ra1, Rb1);
Rc1 := algebra.project(Rc, Ra2);
```

Operator `algebra.select(A,v1,v2)` searches all `key-attr` pairs in base BAT A for attribute values between $v1$ and $v2$. For each qualifying attribute value, the respective `key` value (position) is included in the result. Since selections happen on base BATs, intermediate results are also ordered in the insertion sequence. In MonetDB, intermediate results of selections are simply the keys of the qualifying tuples, thus the positions of where these tuples are stored among the column representations of the relation. In this way, given a key/position we can fetch/project (positional lookup) different attributes of the same relation from their base BATs very fast. Since both intermediate results and base BATs have the attributes ordered in the insertions sequence, MonetDB can very efficiently project attributes by having *cache-conscious reads*.

Operator `algebra.project(A,r)` returns all `key-attr` pairs residing in base BAT A at the positions specified by r. This is a tuple reconstruction operation. Iterating over r, it uses cache-friendly in-order positional lookups into A.

Operators `algebra.KEYintersect(r1,r2)` and `algebra.KEYunion(r1,r2)` are tuple reconstruction operators that perform the conjunction / disjunction of the selection results by returning the intersection / union of keys from r_1 and r_2. Due to order-preserving selection, both r_1 and r_2 are ordered on `key`. Thus, both intersection and union can be evaluated using cache-, memory-, and I/O-friendly sequential data access. The results are ordered on `key`, too, ensuring efficient tuple reconstructions.

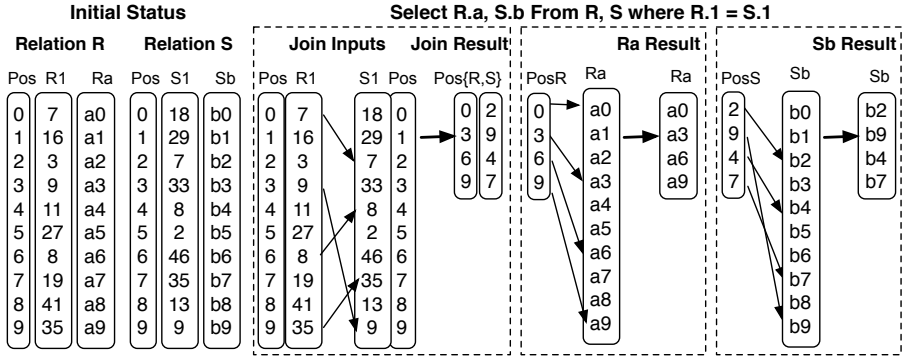


Figure 2.1: Join processing in a column-store

One of the most interesting features of MonetDB is that the actual algorithm used for each operator it is decided at the very last minute as part of the operator call, i.e., a select operator, will on-the-fly decide whether it will do a simple scan select or a binary search if it finds out that the data is sorted. Similarly a join will decide on the fly the proper algorithm depending on the input properties. To a large degree this is one more positive side-effect of the column-based query processing, i.e., processing one column at a time in a bulk mode. This way, we can delay for example the decision of which join algorithm to use up until the moment that we are about to call the join operator. By then, we have fully created the join inputs and thus we can take better decisions.

The join operator in a column-store takes as input two BATs b_1 =(`key1`,`attr1`) and b_2 =(`key2`,`attr2`). The operator finds the joinable `attr1`-`attr2` pairs and produces as output a new BAT that contains the qualifying `key1`-`key2` pairs. These keys are then used in the remainder of the query plan to fetch the qualifying values of the necessary attributes from the base tables and in the correct order. An example of join processing can be seen in Figure 2.1. To find the actual joinable tuples across the two columns, the operator decides on the fly the most appropriate algorithm, e.g., hash join, nested loops join, etc. In addition, the choice of which is considered the inner or outer input is a dynamic one taken at the operator level by exploiting the complete knowledge about the inputs, i.e., their sizes.

The relational operations are grouped into modules, e.g. `algebra`, `aggr` and `bat`. The modules represent a logical grouping and they provide a name space to differentiate similar operations. The actual implementation consists of

several tens of relational primitive implementations, but they are ignored here for brevity. They do not introduce extra complexity either. Each relational operator is a function implemented in C and registered in the database kernel using its extension mechanism. At the moment of writing this thesis, the complete MonetDB package is a sizeable project weighting roughly 2 million lines of C code out of which roughly 250K is the extension required for building the database cracking modules.

Database cracking is implemented within the heart of MonetDB. The cracking implementation represents a set of new operators, algorithms and data structures that cooperate with the existing MonetDB features to give the desired result.

2.11 Summary

In this chapter, we discussed a number of very interesting research areas that have inspired cracking. We argue that the cracking research, as motivated by the results in this thesis, can have a positive impact by bringing a new paradigm and way of understanding database architectures. Of course, the road towards a mature and fully functional system is long when such fundamental changes are introduced. In the next chapters, we introduce in detail database cracking and provide solutions for some of the most fundamental issues, i.e., updates and tuple reconstruction as well as demonstrating further interesting cases where cracking can be exploited, i.e., join processing. In the last chapter, we sketch the bigger picture of the cracking research and potential, touching upon some of the most important research directions that this thesis opens that will bring us even closer to the vision of a truly self-organizing system.

Chapter 3

Selection Cracking*

3.1 Introduction

This chapter presents the basics of database cracking. The simple idea of continuously reorganizing columns based on the query workload is plugged in into a fully functional DBMS resulting to a system prototype that can demonstrate a clear self-organizing behavior. In general, the discussion in this chapter remains at a higher level trying to mainly reflect the basic ideas, intuitions and motivation. Then, the following three chapters go deep into studying and solving analytically specific core research problems of database cracking, demonstrating that it is a viable direction for database architectures but also a very interesting and challenging research path.

3.1.1 Contributions

The contributions of this chapter are the following. We present the first mature cracking architecture, a complete cracking software stack in the context of column-oriented databases. This chapter presents the cracking algorithms that physically reorganize the datastore and the new cracking operators to enable cracking in MonetDB.

Using SQL micro-benchmarks, we assess the efficiency and effectiveness of the system at the operator level. Additionally, we perform experiments that use

*The material in this chapter has been the basis for the CIDR07 paper "Database Cracking" (Idreos et al., 2007a).

the complete software stack, demonstrating that cracker-aware query optimizers can successfully generate query plans that deploy our new cracking operators and thus exploit the benefits of database cracking.

We clearly demonstrate that the resulting system can self-organize according to query workload and that this leads to significant improvement to data access and response times. This behavior is visible even when the user focus is randomly and suddenly shifting to different parts of the data.

3.1.2 Outline

The remainder of this chapter is organized as follows. Section 3.2 discusses database cracking in more detail and introduces the selection cracking architecture. In Section 3.3, we present the algorithms used to perform physical reorganization, while Sections 3.4, 3.5 and 3.6 describe when and how cracking is applied, the new cracking operators and their impact on query plans. In Section 3.7, we provide an evaluation of this initial cracking architecture on top of MonetDB. Finally, Section 3.8 concludes the chapter.

3.2 Selection Cracking

In this section, we introduce the selection cracking architecture. We describe the necessary data structures and give a simple example. It is as follows:

- The *first time* a range query is posed on an attribute A, a cracker database makes a copy of column A. This copy is called the *cracker column* of A, denoted as A_{CRK}.
- A_{CRK} is continuously physically reorganized based on queries that need to touch attribute A.

DEFINITION. Cracking *based on a query q on an attribute A is the act of* physically reorganizing A_{CRK} *in such a way that the values of A that satisfy q are stored in a contiguous space.*

A Simple Example

Consider the example in Figure 3.1. First, query Q1 triggers the creation of cracker column A_{CRK}, i.e., a copy of column A where the tuples are clustered in three pieces, reflecting the ranges defined by the predicate. The result of Q1

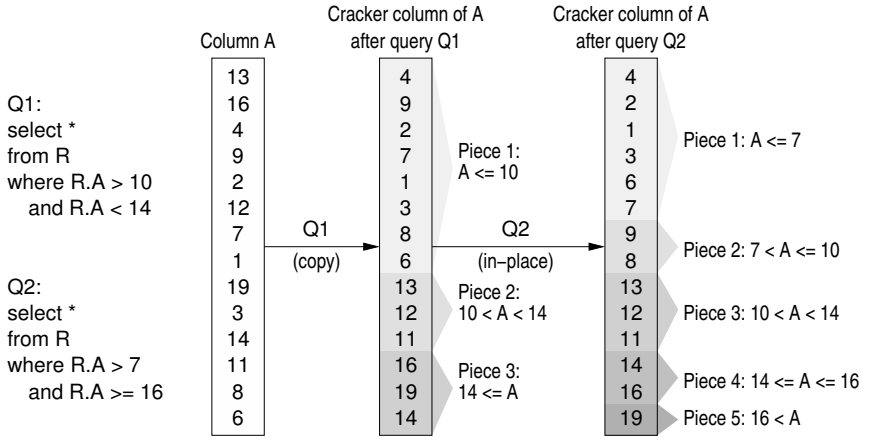


Figure 3.1: Cracking a column

is then retrieved at no extra cost as a *view* on Piece 2 that does not require additional copying of the respective data. We also refer to such views as *column slices*. Later, query Q2 benefits from the information in the cracker index, requiring an in-place refinement of Pieces 1 & 3, only, splitting each in two new pieces, but leaving Piece 2 untouched. The result of Q2 is again a zero-cost column slice, covering the contiguous Pieces 2–4. A third query requesting $A > 16$ would then even exactly match the existing Piece 5.

The Cracker Index

The cracker column is being continuously split into more and more logical pieces as queries arrive. Thus, we need a way to be able to quickly localize a piece of interest in the cracker column. For this purpose, we introduce for each cracker column c a *cracker index*, that maintains information on how values are distributed in c. In our current implementation, the cracker index is an AVL-tree. Each node in the tree holds information for one value v, i.e., it stores a position p referring to the cracker column such that all values that are before position p are smaller than v and all values that are after p are greater. Whether v is left or right inclusive is also maintained.

Data Properties

The way we reorganize data brings structure into the way data is stored. A cracker column consists of multiple logical pieces. Each piece holds values in a given value range while there no overlapping value ranges. The pieces are ordered based on their respective value ranges, i.e., the piece that holds the value range with the smallest values will be physically stored first in the column while the piece with the biggest values will be the last one. Inside each piece, however, there is no order, i.e., values can be in an arbitrary order. Observe Figure 3.1 once more to see these patterns.

Self-organization

The above structure is a dynamic one, continuously changing as more queries arrive. The observation is that by "learning" what a single query can "teach" us, we can speed up multiple queries in the future that request *similar, overlapping or even disjoint* data of the same attribute. With the cracker columns, the system has information about how data is actually stored. We will often refer to this information as *cracking knowledge*. This knowledge can be used to speed up subsequent queries significantly, i.e., queries that request ranges that are an exact match on values known by the index can be answered at the cost of searching the index, only. Even if there is no exact match, the index significantly restricts the values of the column that a query has to analyze. Most importantly, the crucial observation is that this knowledge is dynamic, i.e., the more queries we process, the more we learn, and thus the more we can improve access times.

Challenges

The above introduction of selection cracking naturally raises a number of concerns and questions. For example:

(1) How we reorganize columns?

(2) When does this reorganization happen?

(3) How does cracking affect query plans?

(4) What happens during updates?

(5) What about tuple reconstruction?

(6) Is cracking useful only for selections?

The above questions are just a glimpse of what one may consider given that the cracking ideas affect many basic database design areas. In this chapter, we will answer the first three questions, i.e., we will describe the basic reorganization algorithms, the new operators that apply these algorithms and how these are incorporated in query plans. As the thesis unfolds, we will answer the rest of the questions too, as well as revisiting the choices made in this section where appropriate. Finally, in the last chapter of the thesis, we carve the path of the research challenges raised by database cracking and how it all connects in leading towards a completely self-organizing system.

3.3 How to Crack: The Cracking Algorithms

In this section, we discuss *how* the reorganization happens, i.e., we present the actual algorithms that perform the physical reorganization of a column. The discussion of how these algorithms are incorporated in query processing and when cracking actually happens, comes in subsequent sections.

Cracking Columns

Physical reorganization or cracking is an operation that takes place on an entire column or on a column slice. Two basic cracking operations are needed, called *two-piece* and *three-piece* cracking, respectively. They both have the effect that they physically reorganize a column of an attribute A given a range predicate such as all values of A that satisfy the predicate are in a contiguous space. The former splits the given column into two new pieces while the latter one in three pieces.

Cracking in two pieces uses single-sided predicates, i.e., $A\ \theta\ med$, while cracking in three pieces uses double-sided predicates, i.e., $low\ \theta_1\ A\ \theta_2\ high$, where low, $high$ and med are values in the value range of A and θ, θ_1 and θ_2 are bound conditions.

One could consider even more cracking actions, i.e., four-piece cracking, five-piece cracking, etc. However, the complexity is significantly increased making the creation of an efficient algorithm very hard while at the same time we can use the efficient two- and three-piece cracking to perform any action.

The Algorithms

The algorithms for cracking are formally described in Algorithms 1 and 2. Both algorithms are designed to touch and analyze as little data as possible. The core

Algorithm 1 CrackInTwo(c,posL,posH,med,inc)
Physically reorganize the piece of column c between $posL$ and $posH$ such that all values lower than med are in a contiguous space. inc indicates whether med is inclusive or not, e.g., if $inc = false$ then θ_1 is "<" and θ_2 is ">="

```
1:  x1 = point at position posL
2:  x2 = point at position posH
3:  while position(x1) < position(x2) do
4:    if value(x1) θ1 med then
5:      x1 = point at next position
6:    else
7:      while value(x2) θ2 med &&
        position(x2) > position(x1) do
8:        x2 = point at previous position
9:      exchange(x1,x2)
10:     x1 = point at next position
11:     x2 = point at previous position
```

idea is that, while going through the tuples of a column using 2 or 3 pointers to read, cases where two tuples can be exchanged are carefully identified. The cracking algorithms are cache conscious in the sense that they always try to exploit tuples that are recently read if these tuples must be touched again in the future.

Multiple algorithms were created before ending up with these simple ones. Algorithms that tried to invest in cleverly detecting situations that required fewer exchange operations or allowed early retirement, turned out to be more expensive due to more complex code (e.g., more branches).

We also experimented with "stable" cracking algorithms, i.e., maintain the insertion order of tuples with values in the same range (i.e., tuples that belong in the same piece of the cracker column). However, the algorithms became more complex and two times slower. The fast stable algorithms required a buffer space to temporarily store values subject to move, which is an extra memory overhead.

Cracking Only the Boundary Pieces

The two algorithms presented in this section are sufficient to cover the needs of a column oriented cracking DBMS in terms of physical reorganization. Obviously, the second algorithm that performs three-piece cracking is significantly more

Algorithm 2 CrackInThree(c,posL,posH,low,high,incL,incH) Physically reorganize the piece of column c between $posL$ and $posH$ such that all values in the range low $high$ are in a contiguous space. $incL$ and $incH$ indicate whether low and $high$ respectively are inclusive or not, e.g., if $incL = false$ and $incH = false$ then θ_1 is ">=", θ_2 is ">" and θ_3 is "<"

```
1: x1 = point at position posL
2: x2 = point at position posH
3: while value(x2) θ1 high &&
   position(x2) > position(x1) do
4:    x2 = point at previous position
5: x3 = x2
6: while value(x3) θ2 low &&
   position(x3) > position(x1) do
7:    if value(x3) θ1 high then
8:       exchange(x2,x3)
9:       x2 = point at previous position
10:   x3 = point at previous position
11: while position(x1) <= position(x3) do
12:   if value(x1) θ3 low then
13:      x1 = point at next position
14:   else
15:      exchange(x1,x3)
16:      while value(x3) θ2 low &&
         position(x3)>position(x1) do
17:         if value(x3) θ1 high then
18:            exchange(x2,x3)
19:            x2 = point at previous position
20:         x3 = point at previous position
```

expensive than the two-piece cracking one. It is a more complex algorithm with more pointers, more branches etc.

However, notice that in practice two-piece cracking is the algorithm that is actually used more often. This is the case even when queries use double-sided predicates. Three-piece cracking is used only when *all* the tuples with values in the range requested by a select operator fall into the *same* piece of the cracker column. This is more likely to happen for the first queries that physically reorganize a column. As the column is split into smaller and smaller

pieces by subsequent queries, the chances of requesting a range that falls into only one piece become less and less.

In general, when a value range is requested, it will span over multiple contiguous pieces of a cracker column. However, *only the first and the last piece must be physically reorganized* using two-piece cracking. Recall the example in Figure 3.1. Query Q1 uses a double-sided predicate and since it is the first query that cracks the column it uses three-piece cracking. However, the second query, Q2, that also uses a double-sided predicate does not have to use three-piece cracking. The relevant tuples for Q2 span over Pieces 1, 2 and 3. Cracking does not need to analyze Piece 2 since it is known that all its tuples qualify for the result. Thus, Piece 1 is physically reorganized with two-piece cracking to create two new pieces; one that qualifies for the result and one that does not. Likewise for Piece 3.

Self-organization

Again, these properties add to our quest for self-organization. The more queries touch a given column, i.e., the more the workload converges, the more capable the system becomes of learning more about the data, being able to:

- significantly restrict the data that needs to be analyzed, i.e., it is at most two pieces at a time but the pieces continuously become smaller as more queries arrive.
- run faster cracking algorithms.

3.4 When to Crack: The Operators and Plans

Let us proceed on how we fit the cracking technique in the query plan generator. The crucial point here is the decision of *when* to crack.

Crack While Processing

Cracking is designed towards dynamic environments where we do not expect to have a priori workload knowledge or idle time to prepare the physical design. This way, our decision here is to perform cracking *while* processing queries. This means, that for each query q, the system performs any physical changes, triggered by q, as part of the query processing actions initiated for q, i.e., cracking becomes part of the query plan itself. By the time a query is finished, all cracking actions for this query have been performed.

This is one of the main characteristics of cracking; *lightweight and immediate* reaction to the workload such that the system can very quickly adapt to new patterns and be able to improve performance. At the same time, this is one of the crucial difference of the cracking ideas with the existing techniques on improving the physical design via indices after monitoring and analyzing the workload (see Chapter 2 for a more detailed discussion on cracking vs indices and auto-tuning tools).

Crack Operators

One option would be to process the queries as normal and perform the cracking immediately after each query finishes and before executing the next query. However, this would mean that we would have to load/read the data twice, once for query processing and once for cracking leading in most cases in a significant overhead. Subsequently, given the low level changes introduced by the cracking ideas, our choice is to plug the cracking behavior all the way down to the actual operators.

This way, instead of reacting to the query plan in total, the system automatically reacts on individual operator calls applying the cracking actions as part of the operator steps. In fact, this allows for more flexibility especially as a single query may need access to and require cracking of multiple columns. This will be more evident in the rest of the thesis as we incrementally deal with more and more complex queries, each one requiring to crack several columns with different criteria.

This way, we extend the relational algebra of MonetDB with a small collection of cracker-aware operators. This initial selection cracking architecture is purposely kept as simple as possible in order to understand the implications and the potential. In subsequent chapters, we will see more advanced cracking operators and cracking plans to accommodate more complex queries and bring even more performance improvements.

3.4.1 The `crackers.select` Operator

The first step is to replace the `algebra.select` operator. Recall that the motivation is that cracking happens while processing queries and based on queries. The select operator is typically the main operator that provides access on data and typically feeds the rest of the operators in a query plan. Thus, it is a natural step to choose to experiment with this operator first.

As already discussed in Chapter 2, operators in an advanced column-store with late tuple reconstruction and array based processing need to materialize their result. Here is how a simple select operation works: it receives the column storing a specific attribute, scans it, and creates a new column, containing only tuples with values that satisfy the selection predicate. In our case, in order to explore cracking the select operation will be responsible for the physical reorganization part too. Thus, this new operation should work as follows:

(1) search in the cracker index to determine which piece(s) (at most two) of the cracker column should be touched/physically reorganized

(2) physically reorganize the found piece(s)

(3) update the cracker index if necessary

(4) return the relevant slice of the cracker column as the result (at zero cost).

We extended the MonetDB algebra with the `crackers.select` operator that performs the above steps. Although this may at first seem as if we added additional overhead in the select operation, this is not the case. The fact that the number of tuples to be touched/analyzed is incrementally decreased as more queries crack the column, along with a carefully crafted implementation of the physical reorganization step, leads to an operation orders of magnitude faster than `algebra.select` in MonetDB that needs to scan all tuples in the column for each query. The very first `crackers.select` call on a given column is typically 30% slower than `algebra.select` since it has to reorganize large parts of the column. However, *all* subsequent calls will be faster. As more queries arrive, a select operation on the same attribute becomes cheaper since the cracker index learns more and allows us to touch/analyze smaller pieces of the column (see Section 4.6 for a detailed experimental analysis).

An additional benefit here is the fact that no materialization of the intermediate result is needed; via cracking the qualifying tuples are already in the proper place, i.e., they are clustered in a physically continuous area. This way, we simply need to *point* to this area, avoiding the burden of materializing the result of the operator but still maintaining the benefits of vector-like query processing and late tuple reconstruction.

3.4.2 The `crackers.rel_select` Operator

In the previous section we described the crackers select operator. Let us now see what is the effect of replacing a normal select with a crackers select in a

query plan of MonetDB. Assume the simple select query plan in Section 2.10. This will be replaced with the following cracking plan.

```
Ra1 := crackers.select(Ra, 5, 10);
Rb1 := crackers.select(Rb, 9, 20);
Ra2 := algebra.KEYintersect(Ra1, Rb1);
Rc1 := algebra.project(Rc, Ra2);
```

Although the `crackers.select` individually is much faster, its initial overall effect on the plan turned out to be negligible. The reason is that the `KEYintersect` became more expensive. Recall the description of the basic MonetDB operators in Section 2.10. MonetDB's `algebra.select` produces a result column that is ordered on the insertion sequence of its `key` values. In this way, a subsequent `KEYintersect` can be executed very fast in MonetDB, by exploiting the `key`-order of both operands in a merge-like implementation (i.e., it avoids any random memory access patterns). However, the `crackers.select` returns a column that is no longer ordered on `key`, since it is physically reorganized. This results in a more expensive `KEYintersect` for cracking, requiring a hash-based implementation with inherent random access. For example, experiments with TPC-H query 6 of scale factor 0.1, led to `KEYintersect` being 7 times more expensive.

This side-effect opens a road for detailed studies on both the algebraic operations used and the plan generation scheme of the SQL compiler. An example is the new cracking operator `rel_select`. The goal is to completely avoid the `KEYintersect`. The `rel_select` replaces a pair of a `select` and a `KEYintersect`, performing both simultaneously. It takes an intermediate column c_1 and a base column c_2 as arguments. Due to cracking, c_1 is no longer ordered on `key`. However, being an un-cracked base column, c_2 has a dense sequence of `key` values, stored in ascending order. Thus, iterating over c_1, `rel_select` exploits very fast positional lookup into c_2 to find the matching tuple (c_1.`key` = c_2.`key`), and subsequently checks the selection predicate on c_2.`attr`. The plan transformation was handled readily by the optimizer infrastructure, leading to the following plan for our example.

```
Ra1 := crackers.select(Ra, 5, 10);
Rb1 := crackers.rel_select(Rb, 9, 20, Ra1);
Rc1 := algebra.project(Rc, Rb1);
```

This simple operator allows cracking to materialize its benefits in the overall query cost. In fact, using the `rel_select` can also significantly improve the initial non-cracking MonetDB plans as we will see in our experiments.

This minimal set of just two cracker operators was sufficient to significantly improve the performance of some simple SQL queries and demonstrate that cracking is an interesting research path. In the rest of the thesis, we explore a much larger set of algebraic cracking operators as well as studying in detail individual architectural and algorithmic problems.

3.5 Complexity and Expected Behavior

Having seen the basics of the cracking architecture and algorithms, we can now discuss in more detail about the complexity of cracking a column to help us understand the expected behavior as well.

For ease of presentation, we will first make some simplifying assumptions. Say that each query performs a single two-piece cracking action. In addition, assume that each crack splits a piece of the column into two new pieces of the same size, while each subsequent query symmetrically cracks the column into smaller pieces. For example, the first query will crack the whole column into two equal pieces p_{1a} and p_{1b}, the second query will crack p_{1a} again into two equal pieces p_{2a} and p_{2b}, the third query will crack p_{1b} into two equal pieces p_{2c} and p_{2d}. The fourth query will crack p_{2a}, the fifth query will crack p_{2b} and so on. This simplifies the analysis as now the cost for a sequence of queries cracking a column of N values becomes as follows.

$$N + \frac{N}{2} + \frac{N}{2} + \frac{N}{4} + \frac{N}{4} + \frac{N}{4} + \frac{N}{4} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \ldots$$

The above cost is expressed in terms of data accesses, i.e., how many of the column values the crack algorithm needs to touch/analyze for each query. For example, the very first query needs to analyze every single value in the column and since cracking is a single pass algorithm the cost becomes N data accesses. Given that we assume that we always crack a piece in half, the second query will need to touch $\frac{N}{2}$ values and so on. This way, the cost of the i-th query in such a sequence becomes as follows.

$C_i = \frac{N}{2^{floor(log_2(i))}}$, where $log_2(i) = \frac{log(i)}{log(2)}$.

The important observation here is that with every query that we process, we continuously improve performance as we need to touch less and less data, i.e., the more queries have contributed to cracking the less data a future query needs to analyze.

If we remove our assumptions and consider arbitrary query sequences it is again easy to see that the same patterns hold, i.e., the more queries we have seen in the past, the more chances we have to improve performance in the future simply because we are continuously adding knowledge to the crack columns no matter what is the exact pattern that the queries follow.

Actually, the factor that can be affected by the query patterns is *how fast* performance improves. Depending on the query sequence, performance might improve faster or slower in terms of the number of queries that we need to process before we reach a certain performance level. To capture the notion of expected performance level of a crack column one can think of the amount of knowledge in the column, i.e., the number of logical pieces we have created relative to the number of values/tuples in the column. Of course this can be seen not only for the whole column, but also for subareas, e.g., we might have 200 pieces for a 10 million column where 180 pieces span over the first 1 million tuples and only 20 pieces span over the rest of the tuples. This means that we have more knowledge and thus we can answer more efficiently queries that require ranges in this first part of the column. Of course, this reflects the workload and if the workload shifts we will incrementally get more knowledge for the remaining parts of the column as well.

This way, in a completely random workload each incoming query might be interested for any value range. Given that the value ranges are mapped into areas of the crack columns, this in turn means that queries might need to touch any area of the column spreading over all N values. On the other hand, in a skewed sequence the workload quickly focuses in a specific area of the column of N' tuples where $N' < N$. In this case, cracking quickly restricts data access as less queries will be needed to crack a smaller area to a given performance level as opposed to the whole column in the case of a completely random workload. In both cases, though, the tendency is the same, i.e., continuous and incremental overall improvement by restricting the data the select operator needs to analyze.

We have seen above the expected query cost in a given query sequence. For completeness we should add to this cost the cost of searching the crack index of the column. For the i-th query this is at most $log(i)$, i.e., the depth of the crack index tree. In fact, this cost also represents the lower bound for the cost of a crack select, i.e., when we have an exact hit and thus no actual cracking is necessary as the range requested by this query is already clustered in contiguous area in the column due to past queries.

One of the most natural competitors of cracking is a full sorting strategy, i.e., sort the column when the first query arrives and use binary search from there on. In this case, the first query needs to pay a cost of $N * log(N)$ for the

sorting phase. Every query after that costs only $log(N)$ to perform a typical binary search. This way, cracking is a significantly more lightweight action, i.e., it needs more steps (queries) to get close to the optimal performance of sorting but compared to sorting it has a very lightweight first step and thus it sets the basis for an architecture that quickly reacts to workload changes being able to materialize immediate benefits while this workload is still active.

As described so far cracking purely reacts on queries. However, there are even more opportunities to explore based on the workload and system status. In Chapter 6, we will see how we can "force" auxiliary cracking actions to speed up the learning process in the context of crack joins while Chapter 8 discusses even more alternatives in the context of the big picture of cracking research paths.

3.6 Tuple Reconstruction

Before continuing with the evaluation section, let us first shortly discuss the tuple reconstruction issue. One of our choices in this initial cracking architecture was to create a copy of the column that we are about to crack. This way, the first query has to create the cracker column which of course all subsequent queries can use.

Creating a copy of the column and cracking on it is useful, as it leaves the original column intact where tuples remain in their original order. This order is exploited for positional tuple reconstruction. As we have discussed in Section 2.2, tuple reconstruction is a very crucial issue in column-stores. One of the key ingredients for efficient reconstruction is to exploit positional operations, i.e., exploit the fact that all base columns are aligned. This means that all base columns of a table R hold the values of the $i-th$ in position i. This way, when for example we have a query that selects on an attribute A and then performs an aggregation on an attribute B, it will work as follows. After completing the selection, we have an intermediate result with all the positions of the qualifying A values. Using these positions we can efficiently retrieve the qualifying B values simply by retrieving the respective B values from the same positions in column B.

In this way, if we would reorganize via cracking the base columns, then positional tuple reconstruction would not be possible as the base columns would not be aligned anymore. We would have to perform tuple reconstruction vie expensive join operations. This way, we keep the base columns intact to exploit their tuple order during tuple reconstruction. In fact, in Chapter 5, we are

going to remove these restrictions with the introduction of the more advanced sideways cracking architecture where tuple reconstruction is performed directly on the cracker columns in a self-organizing way.

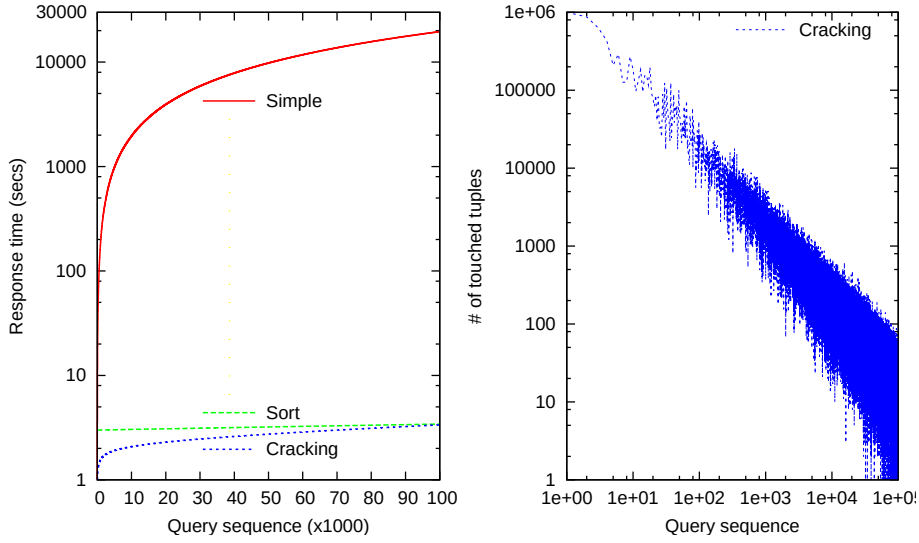
3.7 Experimentation

In this section, we peek into the experimental analysis of our current implementation. The development of cracking was done in MonetDB 5.0alpha and its SQL 2.12 release. All experiments were done on a 2.4 GHz AMD-Athlon 64 with 2 GB memory and a 7200 rpm SATA disk. The experiments are based on a complete implementation. We first discuss a micro-benchmark to assess the individual relational operations. Then, we present results obtained using the complete software stack. Together they provide an outlook on the impact of database cracking.

3.7.1 Select Operator Benchmark

From the large collection of micro-experiments we did to arrive at an efficient implementation, we summarize the behavior of the cracker select against traditional approaches. We tested three select operator variants, (a) simple, (b) sort, and (c) crack, against a series of range queries on a given attribute A. Case (a) uses the default MonetDB select operator, i.e., it scans the column and picks the tuples with values that satisfy the predicate. This is done for every query. Case (b) first performs a sort to physically reorganize the column based on the values of the existing tuples. Then, for each incoming query, it picks the tuples with qualifying values using binary search (e.g., the result is always in a contiguous space in the sorted column). Finally, case (c) uses cracking, i.e., it physically reorganizes (part of) the column for each query. The column has 10^7 tuples (distinct integers from 1 to 10^7). Each query is of the form $v_1 < A < v_2$ where v_1 and v_2 are randomly chosen.

Figure 3.2(a) shows the results of this experiment. On the x-axis queries are ranked in execution order. The y-axis represents the cumulative time for each strategy, i.e., each point (x, y) represents the sum of the cost y for the first x queries. Evidently, cracking and sorting beat the simple scan approach in the long run. A first observation is that, in the long run, both the cracking and the sort strategies are significantly faster than the simple approach since in the latter case we have to scan the whole column for each single query. The simple scan grows linearly with the number of queries. After sorting the data, the cost



(a) Selecting on a 10 million column

(b) Portion touched by crack select

Figure 3.2: Crackers select against the simple and the sort strategy

of selecting any range in the column is in a few micro seconds. The overhead is the sorting phase. The sort loads the first query with an extra cost of 3 seconds, while a simple scan-select needs 0.27 seconds, and the first cracking operation costs 0.38 seconds. For cracking, only the first query is slightly more expensive compared to simple select. All subsequent queries benefit from previous ones and are faster since they do not analyze every column value.

The cost of cracking highly depends on the size of the piece that is being physically reorganized. This is the reason why as more queries come cracking becomes cheaper. This is visible in Figure 3.2(a) by observing the pace with which the cracking curve grows. Initially, the curve grows faster and then as more queries arrive, smaller pieces are cracked and the curve grows with a smaller pace. Cracking has a cost ranging from 10 to 100 micro seconds. Our data show that this variation is due to the size that is cracked each time. Figure 3.2(b) shows the portion of the column that is analyzed each time by the crackers select operator. The more queries arrive, the less data need to be touched/physically reorganized.

Crack vs Sort

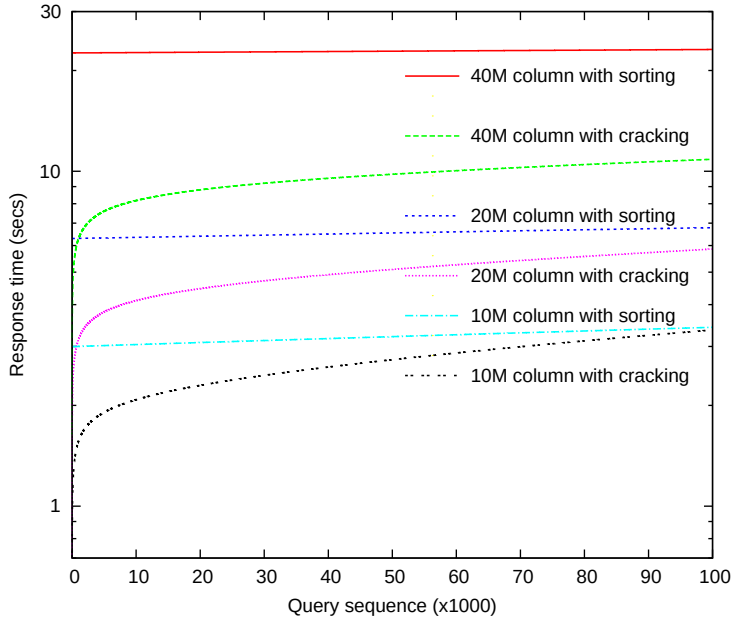
A critical point when comparing cracking with sort is to determine the break-even point, i.e., when their cumulative costs become even. For a 10 M values column in this example, this is at around 10^5 queries in our current implementation (see Figure 3.2(a)). In sort this means that by the time the system answers the first query, cracking has already answered 10^5 queries. Then, the investment of sorting starts to pay off. The overhead of penalizing the first queries however remains, meaning that cracking brings a more normalized behavior.

This is exactly the behavior we want in dynamic environments where we have no knowledge of the workload patterns and no idle time to prepare. For example, sorting here needs 10^5 queries to start materializing any benefits. What if only a couple of relevant queries actually arrive after the very first one that initiated the sorting? Then all the effort to sort the column is a waisted one. Cracking on the other hand, needs only lightweight actions that are hardly visible leading in immediate benefits that can be exploited without delays while the relevant workload pattern is still active. In subsequent chapters, we will discuss more on the crack vs sort topic where we will also see that cracking has the added advantage of being able to handle frequent updates.

Scalability

Figure 3.3 shows how sort and cracking scale on larger columns. As the size increases, cracking becomes more advantageous. For example, observe the points after 10^5 queries to see what cracking gains. This phenomenon can be explained considering the algorithmic complexity. The first cracking operation has a complexity of $O(N)$, where N is the size of the column, while sorting costs $O(N \log N)$.

The results shown in Figure 3.2 are subject to significant improvements by a more "intelligent" maintenance strategy. The break-even point can be shifted further into the future. For example, in Figure 3.2(b) we see that already after the first 8-10 queries cracking touches an order of magnitude less data and becomes significantly faster (10 times faster than simple select). In the experiments so far, the cracker index is always updated and some portion is physically reorganized. As the index is expanded with more knowledge, future searches in the index become more expensive. Cut-off strategies prove effective in this area, i.e., we can disable index updates once the differential cost of subsequent selects drops below a given threshold or we can even drop knowledge on old pieces as we create new ones for the new query patterns. In Chapter 8,



(a) Cracking larger columns

Figure 3.3: Cracking larger columns

we discuss this in more detail.

Selectivity

Here, we study the effect of selectivity on cracking. As before, on a column of 10^7 tuples a series of range queries is fired. This time we vary the selectivity by requesting such ranges that the result size is always S tuples. The area where a query range falls into the value space is still random. The experiment is repeated for S=10^0,10^1,10^2,10^3 and 10^4.

In Figure 3.4, we show the cumulative cost for each run. The main result seen is that *the lower the selectivity the less effort* is needed for cracking to reach optimal performance. This is explained as follows. With higher selectivities cracking creates small pieces in a cracker column to be the result of the current query but leaves large pieces (those outside the result set for the current query)

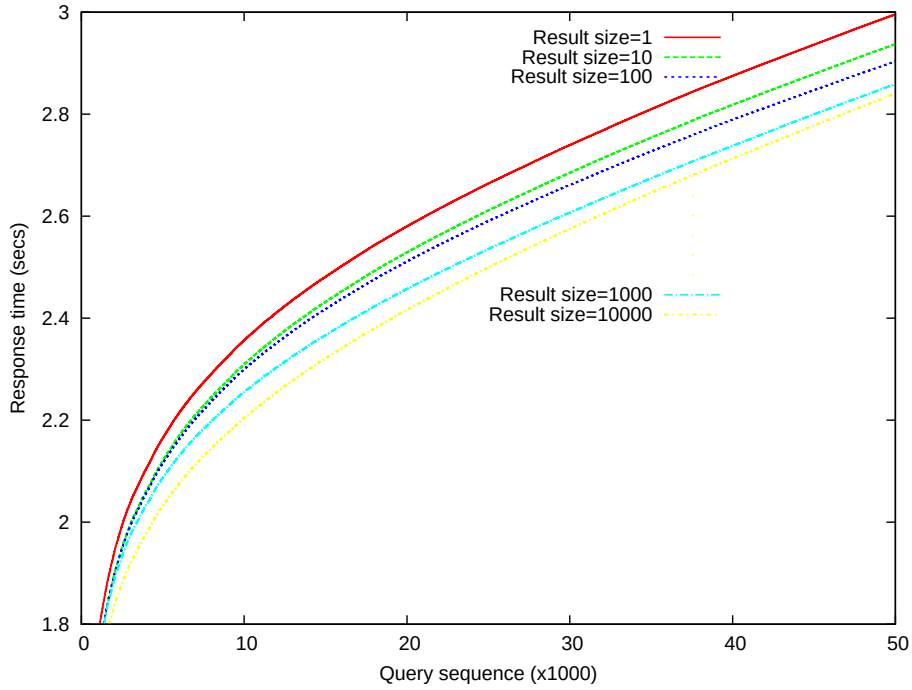


Figure 3.4: Effect of selectivity in the learning process

to be analyzed by future queries. In this way, future queries have a high probability to physically reorganize large pieces. On the contrary, when selecting large pieces the cracker column is partitioned more quickly (with less queries) in more even pieces. However, from Figure 3.4 it is clear that this pattern can be observed but it is not one that dramatically affects performance for cracking. In addition, the fact that ranges requested are random clearly indicates that cracking successfully brings the property of self-organization independently of the selectivities used. In all cases, results obtained outperform those of a sort based or a scan based strategy in a similar way as observed in Figure 3.2.

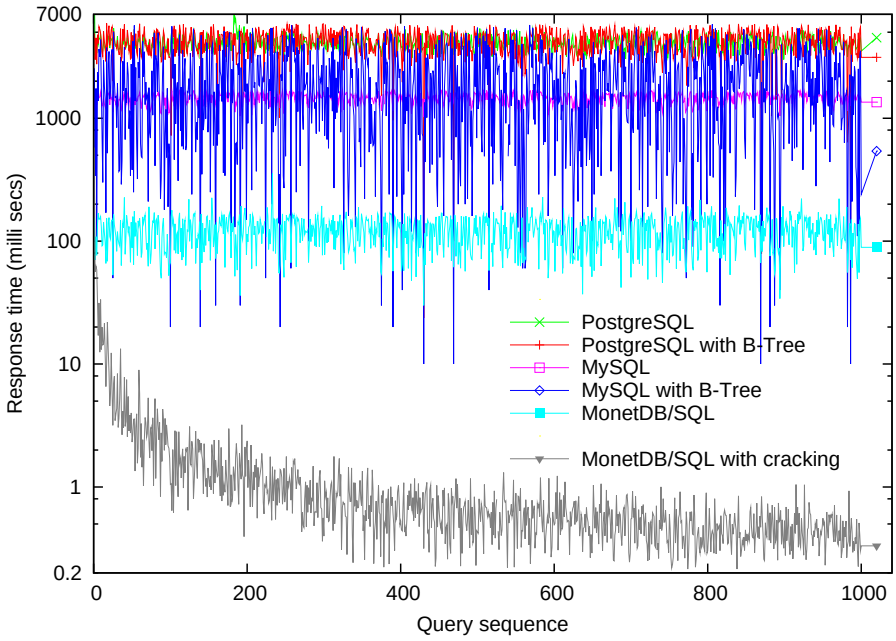


Figure 3.5: A simple count(*) range query

3.7.2 Full Query Evaluation

A novel query processing technique calls for an evaluation in the context of a fully functional system. In this section, we provide an outlook on the evaluation of cracking using SQL queries processed by MonetDB/SQL. The experiments in this section are geared towards demonstrating the potential using some simple SQL queries. More complete analysis will come in subsequent chapters as we deal with specific core research problems of database cracking.

Figure 3.5 shows the results for the following query.

$$\text{select count(*) from R where R.a} > v_1 \text{ and R.a} < v_2$$

The same experiment is ran with PostgreSQL and MySQL both with and without using a B-tree on the attribute. To avoid seeing the DSM/NSM effect, a a single column table is used populated with 10^7 randomly created values be-

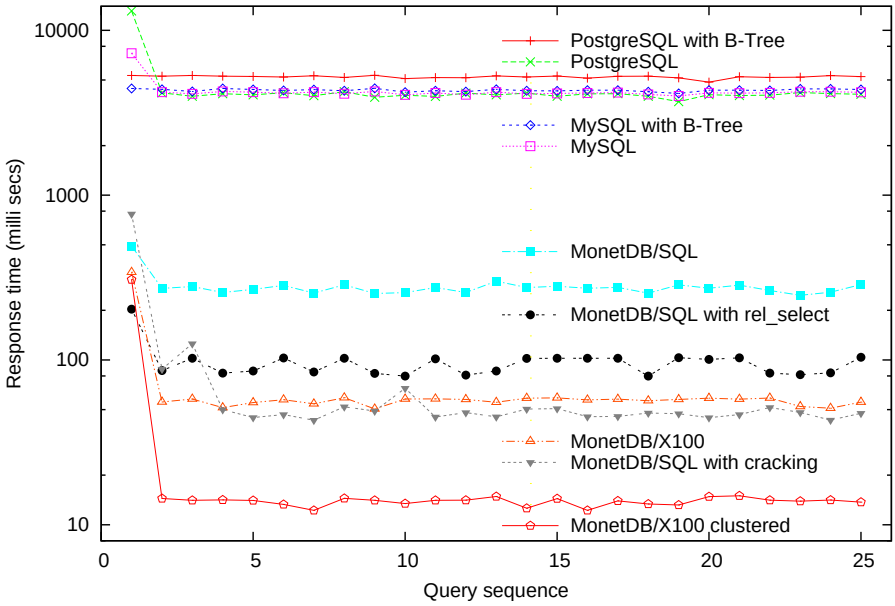


Figure 3.6: TPC-H query 6

tween 0 and 9999. We fire a series of a thousand random queries. Cracking quickly learns and significantly reduces the cost of MonetDB (eventually more than two orders of magnitude) while the rest of the systems maintain a quite stable performance. Since queries are random, selectivities are random. This example clearly demonstrates the ability of cracking to adapt in an environment where there is no up-front knowledge of what queries will be fired. For example, observe that MySQL with a B-tree also reaches high performance for some queries. These are queries with high selectivity where the B-tree becomes useful. However, in order to maintain such performance levels, a traditional system needs up-front knowledge and a stable query workload.

Figure 3.6 shows the results for TPC-H query 6 (a more detailed TPC-H analysis is found in Chapter 5). Again, cracking significantly improves the performance of MonetDB. The graph flattens quickly due to the limited variation in the values of the lineitem shipdate attribute (which is the one that is being cracked). All systems have an extra cost for the first query since this includes

the cost of fetching the data. Since in this query the `rel_select` operator is used, we include a run with the new `rel_select` operator, but without cracking, to clearly show the positive cracking effect. Detailed analysis of the trace shows room for further improvement by exploiting the cracking information in other operators as well.

We also include results obtained with the MonetDB/X100 prototype (Boncz et al., 2005; Zukowski et al., 2006). Its architecture is aimed at pure pipelined. With cracking enabled, MonetDB/SQL performs slightly better than MonetDB/X100 on unclustered data. It can be beaten using a pre-clustered and compressed data ignoring the cost of the initial clustering. This way, it shows the base-line performance achievable in an ideal case. Since the techniques in both source lines are orthogonal, we expect that applying cracking in MonetDB/X100 will have a significant effect as well.

3.8 Summary

In this chapter, we introduced the basic selection cracking architecture. We show that cracking is possible and simple to implement and that changes required in the modules of our experimentation platform MonetDB were straightforward to do. We clearly demonstrate that the resulting system can self-organize based on incoming user requests by significantly restricting data access. The following chapters go in more depth to deal with some of the hard research problems that arise.

Chapter 4

Updates*

4.1 Introduction

Until now, we studied database cracking for the static scenario only, i.e., without updates. A new database architecture should also handle high-volume updates to be considered as a viable alternative. For database cracking, update support is of significant importance. The whole idea of cracking is based on learning from previous queries and continuously physically reorganizing the data. An update, though, changes the physical organization of a column as well. Thus, updates *invalidate* past cracking knowledge given that this knowledge naturally depends on the physical state. Efficiently maintaining the cracking knowledge is the key to make database cracking viable under updates.

Here, we study updates in detail and we show that the nice performance properties of a cracked database can be maintained in a dynamic environment where random updates interleave with random queries.

4.1.1 Contributions

The contributions of this chapter are the following. We present a series of algorithms to support insertions, deletions and updates in a cracking DBMS. We show that our algorithms manage to maintain the advantage of cracking in terms of fast data access. In addition, our algorithms do not hamper the ability of a

*The material in this chapter has been the basis for the ACM SIGMOD07 paper "Updating a Cracked Database" (Idreos et al., 2007b).

cracking DBMS to self-organize, i.e., the system can adapt to query workload with the same efficiency as before and still with no external administration.

The proposed algorithms follow the "cracking philosophy", i.e., with lightweight actions we continuously try to match the current workload in order to have immediate benefits. This way, an update becomes relevant only once a query actually needs the new data. Thus, in cracking, incoming updates are simply marked as pending actions and we update the "cracking" data structures only once queries have to see the updated data. The whole architecture follows the design principles we set in the previous chapter; everything happens on-the-fly *while* processing queries, i.e., updating the cracker columns becomes part of the cracker operators.

The proposed algorithms range from the complete case, where we apply all pending actions in one step, to solutions that update only what is really necessary for the current query; the rest is left for the future when users will become interested in this part of the data.

We implemented and evaluated our algorithms on top of the selection cracking architecture. A detailed experimental evaluation demonstrates that updates can indeed be handled efficiently in a cracking DBMS. A direct comparison with an AVL-tree based scheme highlights the savings obtained with the cracking philosophy. Our study is based on two performance metrics to characterize system behavior. We observe the *total time* needed for a query and update sequence, and our second metric is the *per query response time*. The query response time is crucial for predictability, i.e., ideally we would like similar queries to have a similar response time. We show that it is possible to sacrifice little from the performance in terms of total query sequence cost and to keep the per query response time in a predictable range for all queries.

4.1.2 Outline

The rest of the chapter is organized as follows. In Section 4.2, we discuss how we fitted the update process into the cracking architecture by extending the select operator. Section 4.3 presents a series of algorithms to support insertions in a cracked database. Then, in Section 4.4, we present algorithms to handle deletions, while in Section 4.5 we show how updates are processed. In Section 4.6, we present a detailed experimental evaluation. Finally, Section 4.7 concludes the chapter.

4.2 When to Update: Self-organizing Updates

There are two main issues to consider regarding updates: (a) *when* and (b) *how* updates are applied. Here, we discuss the first issue, postponing the latter to Section 4.3.

What to Update Before we begin our analysis we should mention that updating the base columns is not affected by cracking. Cracking physically changes cracker columns which are copies of the respective base columns. Hence, we assume that an update has already been applied to the base column (i.e., appended) before it has to be applied to the respective cracker column and cracker index. In the remainder of this chapter, we focus on updating the cracking data structures only.

4.2.1 Updating On Demand

If we had enough idle time, we could off-line apply all updates or if we had a priori workload knowledge and enough idle time, we could apply only the most beneficial updates. As we have already discussed in previous chapters though, the cracking vision is to self-organize and adapt to the query workload in dynamic environments where we do not have the luxury of a priori workload knowledge or idle time. And all this should happen without any external human administration leading to a completely autonomous system. Our goal is to *maintain* these properties also in the presence of updates.

One of the key points of the cracking architecture towards this direction is that physical reorganization happens with every query q and while processing q. Each query causes only data relevant for its result to be physically reorganized. Thus, in order to maintain the self-organizing behavior in the presence of updates, the architecture proposed here is in line with the cracking philosophy, i.e., always do *just enough*. Make small investments with every query to gradually adapt to the current workload patterns while these patterns are still active, without disturbing query processing and without introducing any delays.

An update can be seen as a request that becomes relevant only the first time that a query actually needs to analyze the updated data. In cracking, updating the database becomes *part of query execution* in the same way as physical reorganization entered the critical path of query processing.

Let us proceed with the details of our architecture. The cracker columns and indices are *not* immediately updated as updates arrive. Instead, updates are kept in two separate columns for each attribute: the *pending insertions*

column and the *pending deletions* column. When an insert request arrives, the new tuples are simply appended to the relevant pending insertions column. Similarly, the tuples to be deleted are appended in the pending deletions column of the referred attribute. Finally, an update query is simply translated into a deletion and an insertion. Thus, all update operations can be executed very fast, since they result in simple append operations to the pending-update columns.

When a query requests data from an attribute, the relevant cracking data structures are updated *only if necessary*. The updates are applied as part of the crack operators while processing the query. This is a natural extension of the cracking architecture. Recall from previous chapter that the crack select operator makes sure that all values that qualify the given predicate are in a contiguous space in the cracker column. The update-aware select operator does exactly the same thing. To achieve this, it has to physically plug in any relevant updates to the result area.

4.2.2 Update-aware Select Operator

We extended the crack select operator such as it always begins with a check to find out whether there are pending updates for the involved attribute, e.g., values that qualify to be in the result. In this way, only if there are pending update values within the requested value range, then one of our update algorithms runs. Then, the data structures are in a "safe" state so that the cracking select operator can continue normally without losing any information or presenting us information that has been deleted in the past.

To efficiently search in the pending insertions and deletions columns for relevant values, we first sort these columns and then use binary search. We do that because this order will be used by our update algorithms in the future (see next section). All algorithms maintain the order of the pending insertions and deletions columns, so sorting is not necessary in the future, unless new updates arrive.

The exact steps of the operator are as follows:

(1) search the pending insertions column to find qualifying tuples that should be included in the result

(2) search the pending deletions column to find qualifying tuples that should be removed from the result

(3) if at least one of the previous results is not empty, then run an update algorithm

(4) search the cracker index to find which pieces contain the query boundaries

(5) physically reorganize these pieces (at most 2)

(6) return the result.

Steps 1, 2 and 3 are our extension to support updates, while Steps 4, 5 and 6 are the original cracker select operator steps as described in Chapter 3. When the select operator proceeds with Step 4, any pending insertions that should be part of the result have been placed in the cracker column and removed from the pending insertions column. Likewise, any pending deletions that should not appear in the result have been removed form the cracker column and the pending deletions column. Thus, the pending columns continuously shrink when queries consume updates. They grow again with incoming new updates.

Updates are received by the cracker data structures only upon commit, outside the transaction boundaries. By then, they have also been applied to the attribute columns, which means that the pending cracker column updates (and cracker index) can always be thrown away without loss of information. Thus, in the same way that cracking can be seen as dynamically building an index based on query workload, the update-aware cracking architecture proposed can be seen as dynamically updating the index based on query workload.

4.3 Insertions

Let us proceed our discussion on *how* to update the cracking data structures. For ease of presentation, we first present algorithms to handle insertions. Deletions are discussed in Section 4.4 and updates in Section 4.5. We discuss the general issues first, e.g., what is our goal, which data structures do we have to update, how etc. Then, a series of cracker update algorithms are presented in detail.

4.3.1 General Discussion

As discussed in Section 3.2, there are two basic structures to consider for updates in a cracking DBMS, (a) the cracker column and (b) the cracker index. A cracker index I maintains information about the various pieces of a cracker column C. Thus, if we insert a new tuple in any position of C, we have to update the information of I appropriately. We discuss two approaches in detail: one that makes no effort to maintain the index, and a second that always tries to have a valid (cracker-column,cracker-index) pair for a given attribute.

Pending Insertions Column

To comply with the "cracking philosophy", all algorithms start to update the cracker data structures once a query requests values from the pending insertions column. Hence, looking up the requested value ranges in the pending insertions column must be efficient. To ensure this, we sort the pending insertions column once the first query arrives after a sequence of updates, and then exploit binary search. Our merging algorithms keep the pending insertions column sorted. This approach is efficient as the pending insertions column is usually rather small compared to the complete cracker column, and thus, can be kept and managed in memory. We leave further analysis of alternative techniques — e.g., applying cracking with "instant updates" on the pending insertions column — for future research.

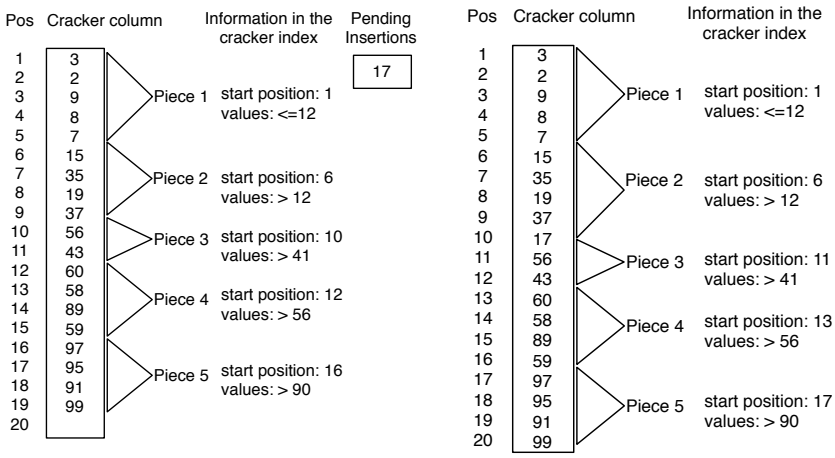
Discarding the Cracker Index

Let us begin with a naive algorithm, i.e., the *forget* algorithm (**FO**). The idea is as follows. When a query requests a value range such that one or more tuples are contained in the pending insertions column, then FO will (a) completely *delete* (forget) the cracker index and (b) simply *append all* pending insertions to the cracker column. This is a simple and very fast operation. Since the cracker index is now gone, the cracker column is again *valid*. From there on, the cracker index is rebuilt from scratch as future queries arrive. The query that triggered FO performs the first cracking operation and goes through all the tuples of the cracker column. The effect is that a number of queries suffer a higher cost, compared to the performance before FO ran, since they will physically reorganize large parts of the cracker column again.

Cracker Index Maintenance

With algorithm FO we delete information (our cracker index) that allows us to answer queries very fast. This has the effect that a number of queries must pay a cost so that we can reach the same levels of performance we had before the update.

Ideally, we would like to handle the appropriate insertions for a given query *without loosing* any information from the cracker index. Then, we could continue answering queries fast without having a number of queries after an update with a higher cost. This is desirable not only because of speed, but also to be able to guarantee a certain level of predictability in terms of response time, i.e., we would like the system to have similar performance for similar queries. This



(a) Before the insertion

(b) After inserting value 17

Figure 4.1: An example of a lossless insertion for a query that requests $5<A<50$

calls for a *merge-like* strategy that "inserts" any new tuple into the correct position of a cracker column and correctly updates (if necessary) its cracker index accordingly.

A simple example of such a "lossless" insertion is shown in Figure 4.1. The left-hand part of the figure depicts a cracker column, the relevant information kept in its cracker index, and the pending insertions column. For simplicity, a single pending insert with value 17 is considered. Assume now a query that requests $5 < A < 50$, thus the pending insert qualifies and should be part of the result. In the right-hand part of the figure, we see the effect of merging value 17 into the cracker column. The tuple has been placed in the second cracker piece, since, according to the cracker index, this piece holds all tuples with value v, where $12 < v \leq 41$. Notice, that the cracker index has changed, too. Information about Pieces 3, 4 and 5 has been updated, increasing the respective starting positions by 1.

Trying to device an algorithm to achieve this behavior, triggers the problem of *moving* tuples in different positions of a cracker column. Obviously, large shifts are too costly and should be avoided. In our example, we moved down by one position all tuples after the insertion point. This is not a viable solution

in large databases. In the rest of this section, we discuss how this merging step can be made very fast by exploiting the cracker index.

4.3.2 Shuffling a Cracker Column

We make the following observation. Inside each piece of a cracker column, tuples have no specific order. This means that a cracker piece p can be shifted z positions down in a cracker column as follows. Assume that p holds k tuples. If $k \leq z$, we obviously cannot do better than moving p completely, i.e., all k tuples. However, in case $k > z$, we can take z tuples from the beginning of p and move them to the end of p. This way, we avoid moving all k tuples of p, but move only z tuples. We will call this technique *shuffling*.

Shuffling From The Top of a Column

For a more complete description, let us first give an example by merging insertions starting from the beginning of a cracker column. Subsequently, we will show that merging from the end of a column is a superior technique.

In the example of Figure 4.1 (without shuffling), 10 tuples are moved down by one position. With shuffling we need to move only 5 tuples. Let us go though the example of Figure 4.1 again, using hopping this time to see why this happens. First, we move v_1, the first value of Piece 3 in a temporary space $temp_1$. The new value 17 can be placed in the position where v_1 was. Then, we move v_2, the first value of Piece 4 in a temporary space $temp_2$. $v1$ will be moved then from $temp_1$ in the position where v_2 was. Then, we move v_3, the first value of Piece 5 at the end of the column. In the position where v_3 was, we move v_2 from $temp_2$. Of course, the cracker index has to be updated so as Pieces 3, 4 and 5 will have start positions greater by one. In this way, we made only 5 moves instead of 10 that we had when moving all values down one position. The benefits of hopping clearly depend on *where* in the cracker column the new value belongs to and also on *how many pieces* exist in the cracker column but in general it is faster than simply shifting all values.

A critical point with shuffling is the *order* in which we merge the updates. As described in the previous paragraph, it requires two temporary spaces to temporarily hold values subject to move. To be safe we have to allocate for *both* spaces enough space to hold as many values as many the values we are trying to insert. Clearly this increases the memory requirements of the algorithm. In addition, having to continuously move values back and forth from the temporary spaces, may seriously hamper performance depending on where in the cracker

column the new values belong and how many pieces exist. Furthermore, the following problem might occur. The values that we have in the temporary space, and that we should put in the begging of the next piece p, might be more than the actual values contained in p. This significantly increases the complexity of correctly updating the cracker index on the starting position of the various pieces since a lot of state has to be kept around. An alternative is to simply choose to delete information about p from the cracker index.

Shuffling From The Bottom of a Column

Our solution to all the above is to change the *direction* of shuffling, i.e., start merging from the *bottom* of the cracker column and not from the piece where the first insertion belongs to. This idea makes everything simple. No state at all has to be kept and no temporary space is needed since there is already free space when we move a value.

Let us go through the example again, this time shuffling from the end of the cracker column to see why. We start from the last piece, Piece 5. The new tuple with value 17 does not belong there. To make room for the new tuple further up in the cracker column, the first tuple of Piece 5, t_1, is moved to the end of the column, freeing its original position p_1 to be used by another tuple. We continue with Piece 4. The new tuple does not belong here, either, so the first tuple of Piece 4 (position p_2), is moved to position p_1. Position p_2 has become free, and we proceed with Piece 3. Again the new tuple does not belong here, and we move the first tuple of Piece 3 (position p_3) to position p_2. Moving to Piece 2, we see that value 17 belongs there, so the new tuple is placed in position p_3 at the end of Piece 2. Finally, the information in the cracker index is updated so that Pieces 3, 4 and 5 have their starting positions increased by one. Thus, only 3 moves were made this time. This advantage becomes even bigger when inserting multiple tuples in one go.

Algorithm 3 contains the details to merge a sorted portion of a pending insertions column into a cracker column. Notice, that the pending insertions column must be sorted for the algorithm to work correctly since insertions are handled one by one starting from the one with the biggest value. In general, the procedure starts from the last piece of the cracker column and moves its way up. In each piece p, the first step is to place at the end of p any pending insertions that belong there. Then, *remaining* tuples are moved from the beginning of p to the end of p. The variable *remaining* is initially equal to the number of insertions to be merged and is decreased for each insertion put in place. The process continues as long as there are pending insertion to merge. If the

Algorithm 3 Merge(C,I,posL,posH)
Merge the cracker column C with the pending insertions column I. Use the tuples of I between positions $posL$ and $posH$ in I.

```
 1: remaining = posH - posL +1
 2: ins = point at position posH of I
 3: next = point at the last position of C
 4: prevPos = the position of the last value in C

 5: while remaining > 0 do
 6:     node = getPieceThatThisBelongs(value(next))
 7:     if node == first piece then
 8:         break

 9:     write = point one position after next
10:     cur = point remaining − 1 positions after write in C

11:     while remaining > 0 and
        (value(ins) > node.value or
        (value(ins) == node.value and node.incl == true))  do
12:         move ins at the position of cur
13:         cur = point at previous position
14:         ins = point at previous position
15:         remaining − −

16:     if remaining == 0 then
17:         break

18:     next = point at position node.position in C
19:     tuples = prevPos - node.position
20:     cur = point one position after next

21:     if tuples > remaining then
22:         w = point at the position of write
23:         copy = remaining
24:     else
25:         w = point remaining − tuples positions after write
26:         copy = tuples

27:     for i = 0; i < copy; i + + do
28:         move cur at the position of w
29:         cur = point at previous position
30:         w = point at previous position

31:     prevPos = node.position
32:     node.position+ = remaining

33: if node == first piece and remaining > 0 then
34:     w = point at position posL
35:     write = point one position after next
36:     for i = 0; i < remaining; i + + do
37:         move cur at the position of w
38:         cur = point at next position
39:         w = point at next position
```

first piece is reached and there are still pending insertions to merge, then all remaining tuples are placed at the end of the first piece. This procedure is the basis for all our merge-like insertion algorithms.

4.3.3 Merge-like Algorithms

Based on the above shuffling technique, we design three merge-like algorithms that differ on the amount of pending insertions they merge per query, ranging from the one extreme of merging all pending insertions in one step, to merging only what is relevant for the current query. All algorithms are based on the same approach regarding *when* they run; an update algorithm will be triggered only if a select operator is called by a query on the given attribute that requests a value range such that *at least one* pending insertion tuple should be part of the result. If no query arrives that requests a value contained in the pending insertions, then the insertions will never be merged. The algorithms also differ in the way they make room for the pending insertions in the cracker column. We continue our discussion by describing each individual merge-like strategy in detail.

MCI

Our first algorithm is called the *merge completely insertions* (**MCI**). algorithm. According to MCI, once a query requests any value from the pending insertions column, the pending insertions column is merged completely, i.e., all pending insertions are placed in the cracker column. The disadvantage is that MCI "punishes" a *single* query with the task to merge all currently pending insertions, i.e., the first query that needs to touch the pending insertions after the new tuples arrived. On the other hand, we are going through the merging process only once so we can save, as we will see in the experiments section, in terms of total execution time.

To run MCI, Algorithm 3 is called for the full size of the pending insertions column. Thus, $posL = 0$ and $posH$ is the size of the pending insertions column. In Figure 4.2(b) we see an example of the result that MCI will have over an attribute that has an initial condition shown in Figure 4.2(a). The query that triggered MCI requests everything between 15 and 50. Since there are such values in the pending insertions, MCI runs. All pending insertions are properly merged, the cracker index has been updated while the pending insertions column is left empty.

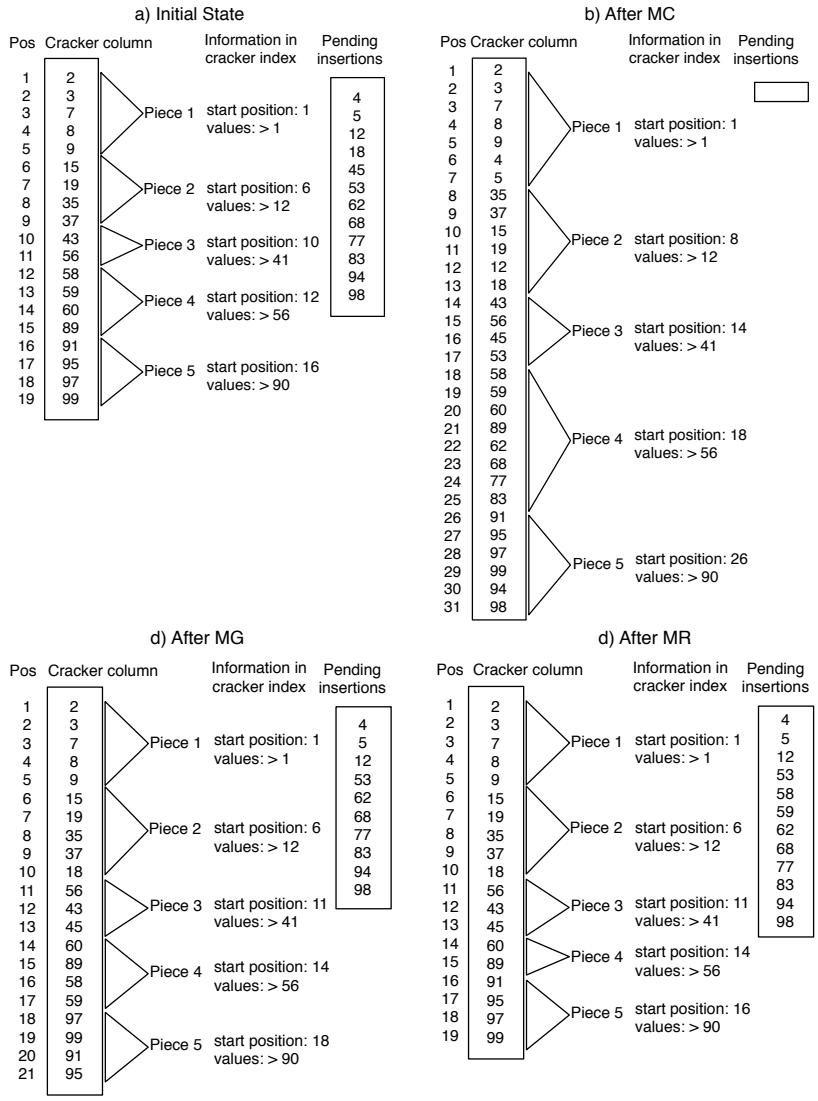


Figure 4.2: An example of how MC, MG and MR will work for a select that requests v such as $15 < v < 50$

MGI

MCI is approaching the cracking philosophy in the sense that it will run only if the result of the current query should contain values that exist in the pending insertions column. Our second algorithm, the *merge gradually insertions*, (**MGI**), algorithm, goes one step further. In MGI, if a query needs to touch k tuples from the pending insertions column, where $k \geq 1$, then it will merge *only these* k tuples into the cracker column, and not all pending insertions. The remaining pending insertions wait for future queries to consume them. Thus, MGI does not burden a single query to merge all pending insertions.

For MGI, Algorithm 3 runs for only a portion of the pending insertions column that qualifies as query result. When this procedure is finished, MGI needs to shrink the pending insertions column so that it is dense again and can be reused in the future. The exact positions are easy/cheap to find with binary search since the pending insertions column is sorted. This is done by shifting up the values that did not qualify for the given select and are after the consumed insertions. In Figure 4.2(c) we show an example of what the result of MGI will be. We see that only the necessary values for the current query are merged, values 18 and 45 so the cracker column grew only two positions. In addition, MGI has shrank the pending insertions column by two position since 18 and 45 need no longer to be there. We will show in the experiments section that although MGI outperforms MCI in terms of cost per query, at the same time this comes with a high total cost, i.e., the total time of processing a sequence of queries and insertions becomes very expensive.

MRI

Our third algorithm is called the *merge ripple insertions* (**MRI**) algorithm. The basic idea behind MRI is triggered by the following observation about MCI and MGI. In general, there is a number of pieces in the cracker column that we shift down by shuffling until we start merging. These are all the pieces from the end of the column until the piece p_h where the tuple with the *highest* qualifying value belongs to. These pieces are *irrelevant* for the current query since they are outside the desired value range. All we want, regarding the current query, is to make enough room for the insertions we must merge. This is exactly why we shift these pieces down.

To merge k values MRI starts directly at the position that is after the last tuple of piece p_h. From there, k tuples are moved into a temporary space *temp*. Then, the procedure of Algorithm 3 runs for the qualifying portion of

the pending insertions as in MGI. The only difference is that now the procedure starts merging from piece p_h and not from the last piece of the cracker column. Finally, the tuples in *temp* are merged into the pending insertions column. Merging these tuples back in the cracker column is left for future queries. Note, that for a query q, all tuples in *temp* have values greater than the pending insertions that had to be merged in the cracker column because of q (since these tuples are taken from after piece p_h). This way, the pending insertions column is continuously filled with tuples with *increasing* values up to a point where we can simply *append* these tuples at the cracker column without affecting the cracker index (i.e., tuples that belong to the last piece of the cracker column).

Let us go through the example of Figure 4.1 again, using MRI this time. Piece 3 contains the tuple with the highest qualifying value. We have to merge tuple t with value 17. The tuple with value 60 is moved from position 12 in the cracker column to a temporary space. Then the procedure of Algorithm 3 starts from Piece 3. t does not belong in Piece 3 so the tuple with value 56 is moved from position 10 (the first position of Piece 3) to position 12. Then, we continue with Piece 2. t belongs there so it is simply placed in position 10. The cracker index is also updated so that Pieces 3 and 4 have their starting positions increased by one. Finally, the tuple with value 60 is moved from the temporary space to the pending insertions. At this point MRI finishes without having shifted Pieces 4 and 5 as MCI and MGI would have done.

In Section 4.6, a detailed analysis is provided that clearly shows the advantage of MRI by avoiding the *unnecessary* shifting of non-interesting pieces. Of course, the performance of all algorithms highly depends on the scenario, e.g., how *often* updates arrive, how *many* of them and how often queries ask for the values used in the new tuples. We examine various scenarios and show that all merge-like algorithms always outperform the non-cracking and AVL-case.

4.4 Deletions

Deletion operations form the counter-part of insertions and they are handled in the same way, i.e., when a new delete query arrives to delete a tuple d from an attribute A, it is simply appended to the pending deletions column of A. Only once a query requests tuples of A that are listed in its pending deletions column, d might be removed from the cracker column of A (depending on the delete algorithm used). Our deletion algorithms follow the same strategies as with insertions; for a query q, (a) the *merge completely deletions* (**MCD**) removes *all* deletions from the cracker column of A, (b) the *merge gradually deletions*

(**MGD**) removes only the deletions that are relevant for q and (c) the *merge ripple deletions* (**MRD**), similar to MRI, touches only the relevant parts of the cracker column for q and removes only the pending deletions interfering with q.

Let us now discuss *how* pending deletes are removed from a cracker column C. Assume for simplicity a single tuple d that is to be removed from C. The cracker index is again used to find the piece p of C that contains d. For insertions, we had to make enough space so that the new tuple can be placed in *any* position in p. For deletions we have to *spot* the position of d in p and clear it. When deleting a single tuple, we simply scan the (usually quite small) piece to locate the tuple. In case we need to locate multiple tuples in one piece, we apply a join between the piece and the respective pending deletes, relying on the underlying DBMS's ability to evaluate the join efficiently e.g., by building a (temporary) hash table on the fly.

We do that by performing a join operation as follows. We create a non materialized view (slice) Cs of p. We do the same for the portion of D where deletions for p lay. Then we have Ds. Both slices are binary tables of type (key-value). In Cs we replace all values in the tail with increasing numbers starting from 0 (that denote the positions of tuples in p). Then, we reverse Cs so that the tail becomes the head and the head the tail. Then we get the positions of deletes in p by a relational join between Cs and Ds. The algorithm used for the join is handled by the MonetDB kernel depending on the sizes of the two views.

Once the position of d is known, it can be seen as a "hole" which we must fill to adhere to the data structure constraints of the underlying DBMS kernel. We simply take a tuple from the end of p and move it to the position of d, i.e., we use shuffling to *shrink* p. This leads to a hole at the end of p. Consequently, all subsequent pieces of the cracker column need to be *shifted up* using shuffling. Thus, for deletions the merging process starts from the piece where the lowest pending delete belongs to and *moves down* the cracker column. This is the opposite of what happens for insertions, where the procedure *moves up* the cracker column. Conceptually, removing deletions can also be seen as moving holes down until all holes are at the end of the cracker column (or at the end of the interesting area for the current query in the case of MRD), where they can simply be ignored.

In MRD, the procedure stops when it reaches a piece where all tuples are outside the desired range for the current query. Thus, *holes will be left* inside the cracker column *waiting for future queries* to move them further down, *if needed*. In Algorithm 4, we formally describe MRD. Variable *deletions* is initially equal to the number of deletes to be removed and is increased if holes are found inside the result area, left there by a previous MRD run. The algorithm for MCD and

Algorithm 4 RippleDeletions(C,D,posL,posH, low, incL, hgh, incH)
Update C with the tuples between positions $posL$ and $posH$ of D.

```
1: remaining = posH - posL +1
2: del = point at first position of D
3: Lnode = getPieceThatThisBelongs(low, incL)
4: stopNode = getPieceThatThisBelongs(hgh, incH)
5: LposDe = 0

6: while true do
7:     Hnode = getNextPiece(Lnode)
8:     delInCurPiece = 0
9:     while remaining > 0 and (value(del) > Lnode.value or
       (value(del) == Lnode.value and Lnode.incl == true)) and (value(del) > Hnode.value or
       (value(del) == Hnode.value and Hnode.incl == true))  do
10:        del = point at next position
11:        delInCurPiece ++
12:    LposCr = Lnode.pos + (deletions − remaining)
13:    HposCr = Hnode.pos
14:    holesInCurPiece = Hnode.holes
15:    if delInCurPiece > 0 then
16:        HposDe = LposDe + delInCurPiece
17:        positions = getPos(b, LposCr, HposCr, u, LposDe, HposDe)
18:        pos = point at first position in positions
19:        posL = point at last position in positions
20:        crk = point at position HposCr in C
21:        while pos <= posL do
22:            if position(posL)! = position(crk) then
23:                copy crk into pos
24:                pos = point at next position
25:            else
26:                posL = point at previous position
27:            crk = point at previous position
28:    holeSize = deletions − remaining
29:    tuplesInCurPiece = HposCr − LposCr − delInCurPiece
30:    if holeSize > 0 and tuplesInCurPiece > 0 then
31:        if holeSize >= tuplesInCurPiece then
32:            copy tuplesInCurPiece tuples from position (LposCr + 1)
33:            at position (LposCr − (holeSize − 1))
34:        else
35:            copy holeSize tuples from position
36:            (LposCr + 1 + (tuplesInCurPiece − holeSize))
37:            at position (LposCr − (holeSize − 1))
38:    if tuplesInCurPiece == 0 then Lnode.deleted = true
39:    remaining− = delInCurPiece
40:    deletions+ = holesInCurPiece
41:    if Hnode == stopNode then break
42:    LposDe = HposDe
43:    Hnode.holes = 0
44:    Lnode = Hnode
45:    Hnode.pos− = holeSize + delInCuPiece + holesInCurPiece

46: if highNode == last piece then C.size− = (deletions − remaining)
47: else Hnode.holes = deletions − remaining
```

MGD is similar. The difference is that it stops only when the end of the cracker column is reached.

For MRD, we need more administration. For every piece p in a cracker column, we introduce a new variable (in its cracker index) to denote the number of holes before p. We also extend the update-aware select operator with a 7th step that removes holes from the result area, if needed. Assume a query that does not require consolidation of pending deletions. It is possible that the result area, as returned by step 6 of the update-aware cracker select, contains holes left there by previous queries (that ran MRD). To remove them, the following procedure is run. It starts from the first piece of the result area P in the cracker column and steps down piece by piece. Once holes are found, we start shifting pieces up by shuffling. The procedure finishes when it is outside P. Then, all holes have been moved to the end of P. This is a simplified version of Algorithm 4 since here there are no tuples to remove.

4.5 Updates

A simple way to handle updates is to translate them into deletions and insertions, where the deletions need to be applied before the respective insertions in order to guarantee correct semantics.

However, since our algorithms apply pending deletions and insertions (i.e., merge them into the cracker column) purely based on their attribute *values*, the correct order of deletions and insertions of the same *tuples* is not guaranteed by simply considering pending deletions before pending insertions in the update-aware cracker select operator. In fact, problems do not only occur with updates, but also with a mixture of insertions and deletions. Consider the following three cases.

(1) A recently inserted tuple is deleted before the insertion is applied to the cracker column, or after the inserted tuple has been re-added to the pending insertions column by MRI. In either case, the same tuple (identical key and value) will appear in both the pending insertions and the pending deletions column. Once a query requests (the attribute value of) that tuple, it needs to be merged into the cracker column. Applying the pending delete first will not change the cracker column, since the tuple is not yet present there. Then, applying the pending insert, will add the tuple to the cracker column, resulting in an incorrect state. We can simply avoid the problem by ensuring that a to-be-deleted tuple is not appended to the

pending deletions column, if the same tuple is also present in the pending insertions column. Instead, the tuple must then be removed from the pending insertions column. Thus, the deletion effectively (and correctly) cancels the not yet applied insertion.

(2) The same situation occurs if a recently inserted (or updated) tuple gets updated (again) before the insertion (or original update) has been applied. Again, having deletions cancel pending insertions of the same tuple with the same value solved the problem.

(3) A similar situation occurs, when MRI re-adds "zombie" tuples, a pending deletion which has not yet been applied, to the pending insertions column. Here, the removal of the to-be-deleted tuple from the cracker column implicitly applies the pending deletion. Hence, the respective tuple must not be re-added to the pending insertions column, but rather removed from the pending deletions column.

In summary, we can guarantee correct handling of interleaved insertions and deletions as well as updates (translated into deletions and insertions), by ensuring that a tuple is added to the pending insertions (or deletions) only if the same tuples (identical key and value) does not yet exist in the pending deletions (or insertions) column. In case it does already exist there, it needs to be removed from there.

This scheme is enough to efficiently support updates in a cracked database without any loss of the desired cracking properties and speed. Our future work plans include research on unified algorithms that combine the actions of merging pending insertions and removing pending deletions in one step for a given cracker column and query. Such algorithms could potentially lead to even better performance.

4.6 Experimental Analysis

In this section, we demonstrate that our algorithms allow a cracking DBMS to maintain its advantages under updates. This means that queries can be answered faster as time progress and we maintain the property of self-adjustment to query workload. The algorithms are integrated in the MonetDB code base.

All experiments are based on a single column table with 10^7 tuples (unique integers in $[1, 10^7]$) and a series of 10^4 range queries. The range always spans 10^4 values around a randomly selected center (other selectivity factors follow). We

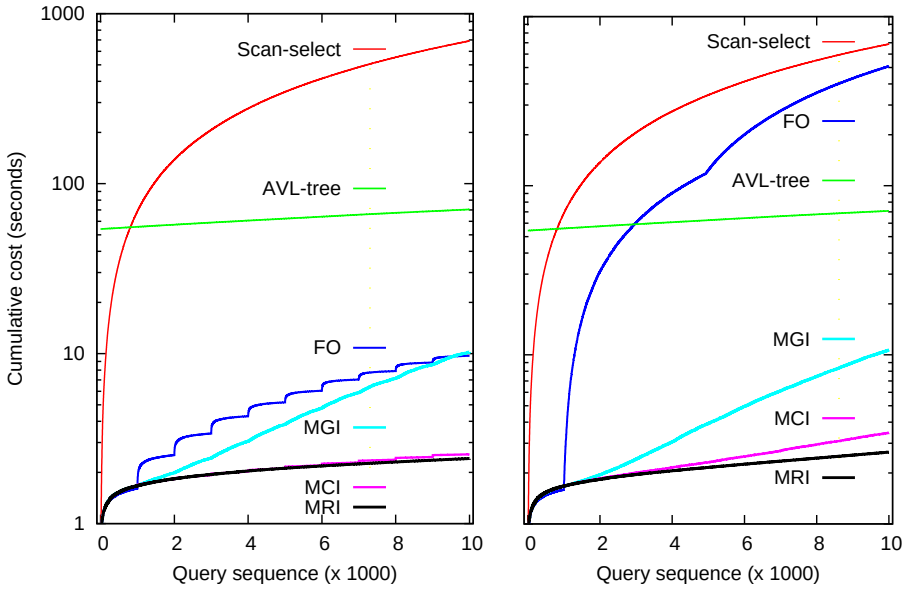
study two update scenarios, (a) low frequency high volume updates (LFHV), and (b) high frequency low volume updates (HFLV). In the first scenario batch updates containing a large number of tuples occur with large intervals, i.e., many queries arrive between updates. In the second scenario, batch updates containing a small number of tuples happen more often, i.e., only a small number of queries have arrived since the previous updates. In all LFHV experiments we use a batch of 10^3 updates after every 10^3 queries, while for HFLV we use a batch of 10 updates after every 10 queries. Update values are randomly chosen in $[1, 10^7]$.

All experiments are conducted on a 2.4 GHz AMD Athlon 64 processor equipped with 2 GB RAM and two 250 GB 7200 rpm S-ATA hard disks configured as software-RAID-0. The operating system is Fedora Core 4 (Linux 2.6.16).

4.6.1 Basic Insights

For readability, we start with insertions to obtain a general understanding of the algorithmic behavior. We compare the update-aware cracker select operator against the scan-select operator of MonetDB and against an AVL-tree index created on top of the columns used. To avoid seeing the "noise" from cracking of the first queries we begin the insertions after a thousand queries have been handled. For example, for LFHV the first insertions arrive after a thousand queries and then after every thousand queries, while for the HFLV scenario the first insertions arrive after a thousand queries and then every ten queries. Figure 4.3 shows the results of this experiment for both LFHV and HFLV. The x-axis ranks queries in execution order. The logarithmic y-axis represents the cumulative cost, i.e., each point (x, y) represents the sum of the cost y for the first x queries. The figure clearly shows that all update-aware cracker select algorithms are superior to the scan-select approach. The scan-select scales linearly, while cracking quickly adapts and answers queries fast. The AVL-tree index has a high initial cost to build the index, but then queries can be answered fast too. For the HFLV scenario, FO is much more expensive. Since updates occur more frequently, it has to forget the cracker index frequently, restarting from scratch with only little time in between updates to rebuild the cracker index. Especially with MCI and MRI, we have maintained the ability of the cracking DBMS to reduce data access.

Notice, that both the ranges requested and the values inserted are randomly chosen, which demonstrates that all merge-like algorithms maintain the ability of a cracking DBMS to self-organize and adapt to query workload.



(a) LFHV scenario (b) HFLV scenario

Figure 4.3: Cumulative cost for insertions

In the LFHV scenario (Figure 4.3 (a)) algorithms FO, MCI, MGI and MRI are all orders of magnitude faster than the normal select. For example, if we look at the total cost required for the 10^4 queries to run (this is the point where the curves cross the right y-axis), FO and MGI are two orders of magnitude faster than the normal select while MCI and MRI are almost three orders of magnitude faster.

To discuss in more detail the various issues, we will use Figure 4.4. Figure 4.4 shows the cost per query through the complete LFHV scenario sequence. The scan-select has a stable performance at around 80 milliseconds while the AVL-tree has a high initial cost to build the index, but then query cost is never more than 3.5 milliseconds. When more values are inserted into the index, queries cost slightly more. Again FO behaves poorly. Each insertion incurs a higher cost to recreate the cracker index. After a few queries performance becomes as good as it was before the insertions.

MCI overcomes the problem of FO by merging the new insertions only when

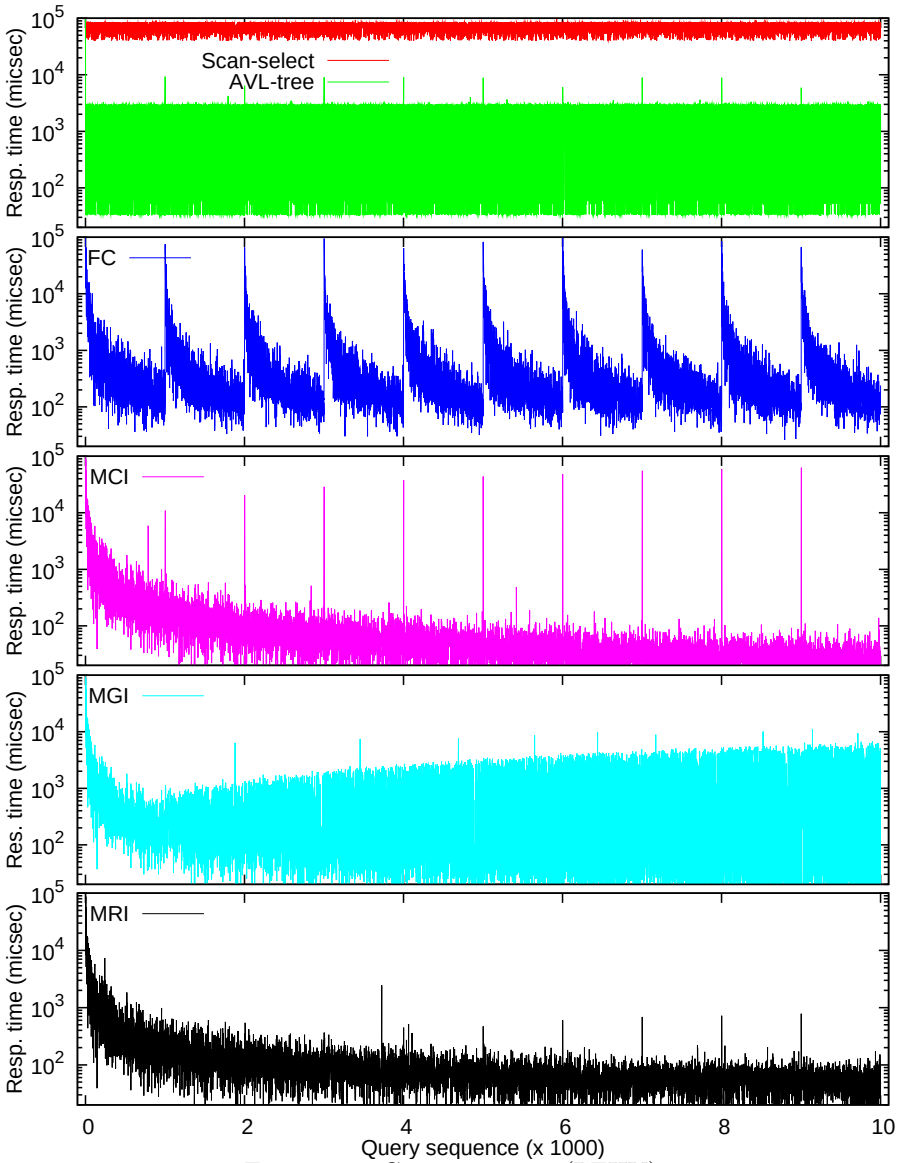
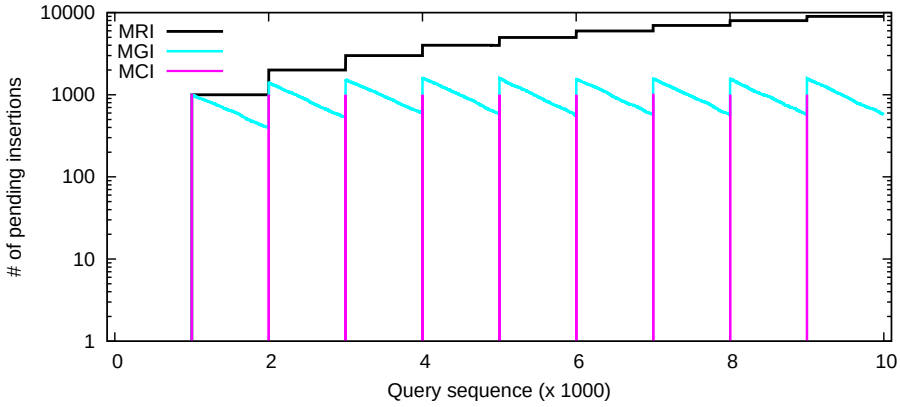
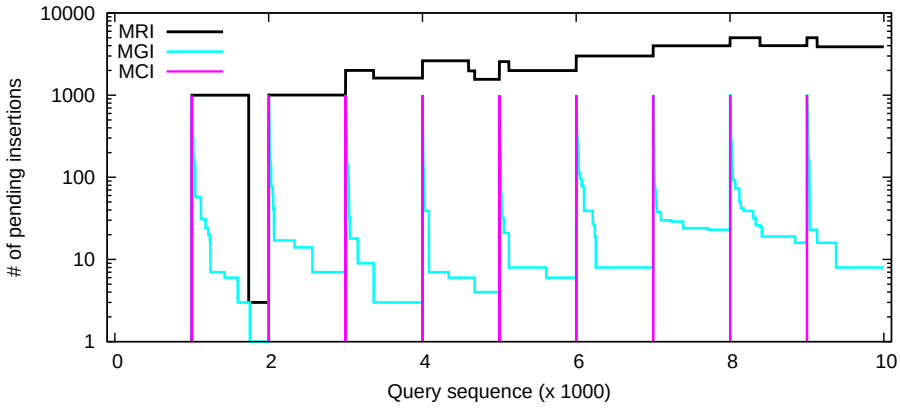


Figure 4.4: Cost per query (LFHV)



(a) Result size 10^4 values



(b) Result size 10^6 values

Figure 4.5: Number of pending insertions (LFHV)

requested for the first time. *A single* query suffers extra cost after each insertion batch. Moreover, MCI performs a lot better than FO in terms of total cost as seen in Figure 4.3, especially for the HFLV scenario. However, even MCI is problematic in terms of cost per query and predictability. The first query interested in one or more pending insertions suffers the cost of merging all of them and gets an exceptional response time. For example, a few queries carry a

response time of ca. 70 milliseconds, while the majority cost no more than one millisecond.

Algorithm MGI solves this issue. All queries have a cost less than 10 milliseconds. MGI achieves to balance the cost per query since it always merges fewer pending insertions than MCI, i.e., it merges only the tuples required for the current query. On the other hand, by not merging all pending insertions, MGI has to merge these tuples in the future when queries become interested. Going through the merging process again and again causes queries to run slower compared to MCI. This is reflected in Figure 4.3, where we see that the total cost of MGI is a lot higher than that of MCI.

MRI improves on MGI because it can avoid the very expensive queries. Unlike MGI it does not penalize the rest of the queries with an overhead. MRI performs the merging process only for the interesting part of the cracker column for each query. In this way, it touches less data than MGI (depending on where in the cracker column the result of the current query lays). Comparing MRI with MCI in Figure 4.4, we see the absence of very expensive queries, while comparing it with MGI, we see that queries are much cheaper. In Figure 4.3, we also see that MRI has a total cost comparable to that of MCI.

In conclusion, MRI performs better than all algorithms since it can keep the total cost low without having to penalize a few queries. Performance in terms of cost per query is similar for the HFLV scenario, too. The difference is that for all algorithms the peaks are much more frequent, but also lower, since they consume fewer insertions each time. We present a relevant graph later in this section.

4.6.2 Effect of the Number of Pending Insertions

To deepen our understanding on the behavior of the merge-like algorithms, we measure in this experiment the number of pending insertions left after each query has been executed. We run the experiment twice, having the requested range of all queries span 10^4 and 10^6 values, respectively.

In Figure 4.5, we see the results for the LFHV scenario. For both runs, MCI insertions are consumed very quickly, i.e., only a few queries after the insertions arrived. MGI continuously consumes more and more pending insertions as queries arrive. Finally, MRI keeps a high number of pending insertions since it replaces merged insertions with tuples from the cracker column (unless the pending insertions can be appended). For the run with the lower selectivity we observe for MRI that the size of the pending insertions is decreased multiple times through the query sequence which means that MRI had the chance to

simply append pending insertions to the cracker column.

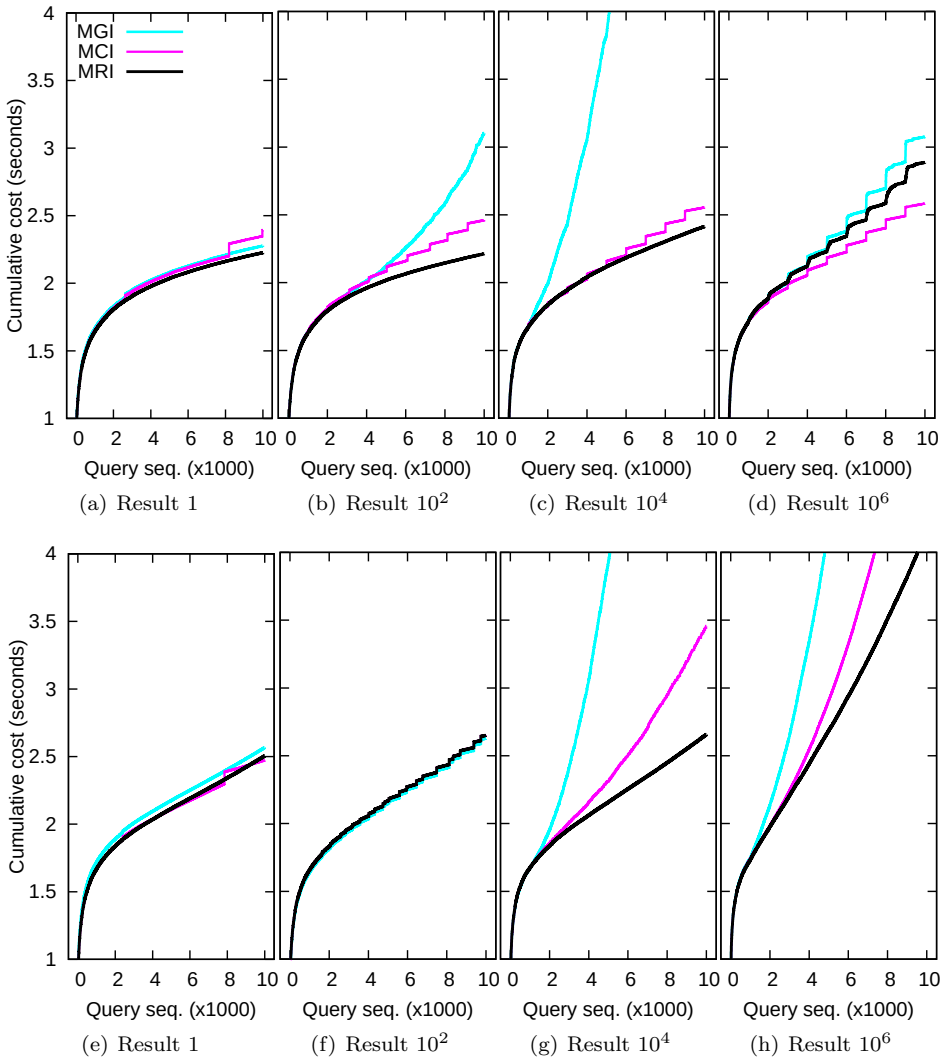
4.6.3 Selectivity effect

Having sketched the major algorithmic differences of the merge-like update algorithms and their superiority compared to the non-cracking case, we discuss here the effect of selectivity. First, the algorithms are triggered only when the result of a query must contain part of the pending insertions. Obviously, the lower the selectivity of the query (i.e., the larger the result size) the higher the probability of this to happen. In addition, MGI and MRI merge only the values that are necessary for the current query so again selectivity can affect their performance.

For this experiment, we fire a series of 10^4 random range queries that interleave with insertions as before. However, different selectivity factors are used such that the range spans over (a) 1 (point queries), (b) 100, (c) 10^4 and (d) 10^6 values.

In Figure 4.6, we show the cumulative cost. Let us first discuss the LFHV scenario. For point queries we see that all algorithms have a quite stable performance. With such a high selectivity, the probability of requesting a tuple from the pending insertions is very low. Thus, most of the queries do not need to touch the pending insertions, leading to a very fast response time for all algorithms. Only MCI has a high step towards the end of the query sequence, caused by a query that needs one tuple from the pending insertions, but since MCI merges all insertions, the cost of this query becomes high. As the selectivity drops, all update algorithms need to operate more often. Thus, we see higher and more frequent steps in MCI. For MGI observe that initially, as the selectivity drops, the total cost is significantly increased. This is because MGI has to go though the update process very often by merging a small number of pending insertions each time. However, when the selectivity becomes even lower, e.g., 1/10 of the column, MGI again performs well since it can consume insertions faster. Initially, with a high selectivity, MRI is faster in total than MCI but with dropping selectivity it looses this advantage due to the merging process being triggered more often. The difference in the total cost when selectivity is very low, is the price to pay for having a more balanced cost per query. MCI loads a number of queries with a high cost which is visible in the steps of the MCI curves. In MRI curves, such high steps do not exist.

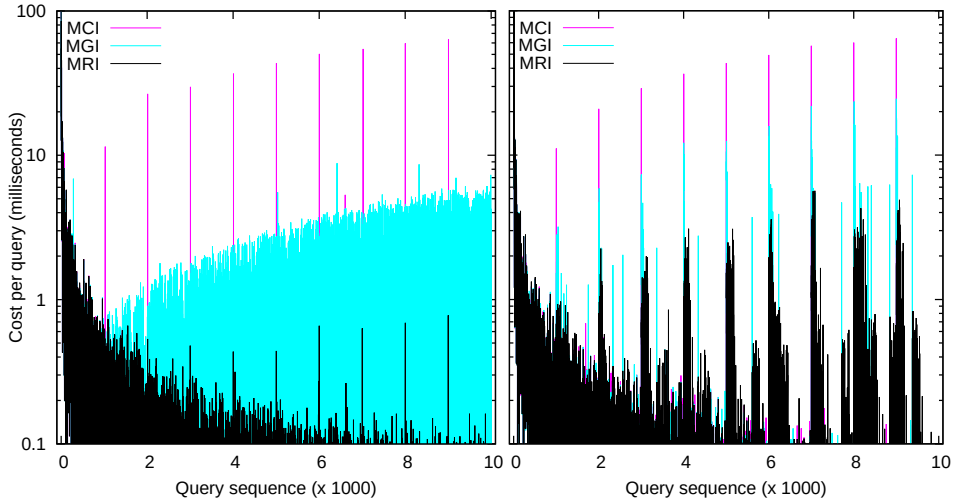
For the HFLV scenario, MRI always outperforms MCI. The pending insertions are consumed in small portions very quickly since they occur more often. In this way, MRI avoids doing expensive merge operations for multiple values.



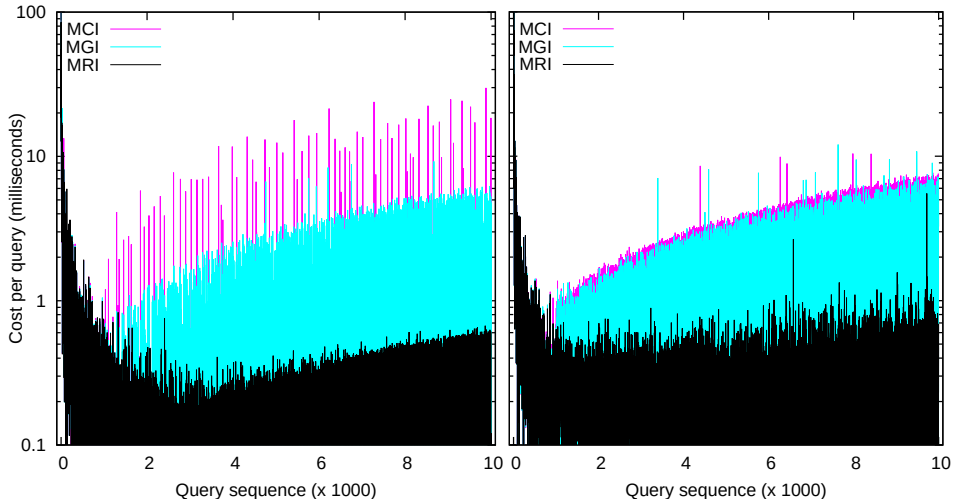
(a) Result 1 (b) Result 10^2 (c) Result 10^4 (d) Result 10^6

(e) Result 1 (f) Result 10^2 (g) Result 10^4 (h) Result 10^6

Figure 4.6: Effect of selectivity in cumulative cost in the LFHV (a,b,c,d) and in the HFLV (e,f,g,h) scenario



(a) Result size 10^3 values in LFHV scenario (b) Result size 10^6 values in LFHV scenario



(c) Result size 10^3 values in HFLV scenario (d) Result size 10^6 values in HFLV scenario

Figure 4.7: Effect of selectivity in per query cost

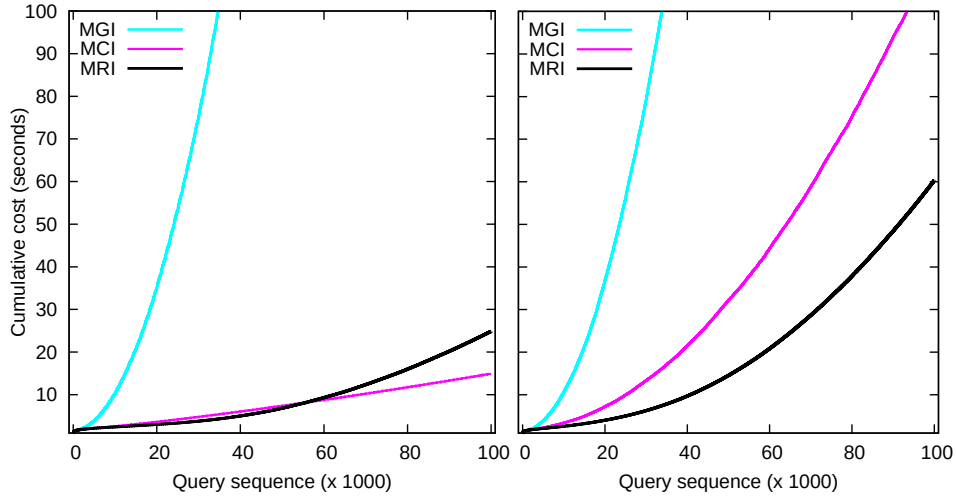
In Figure 4.7, we illustrate the cost per query for a low and a high selectivity for both scenarios. In general, the same pattern as in our first experiment can be observed. MRI maintains its advantage in terms of not penalizing single queries.

Let us first discuss the LFHV scenario. As we see in Figure 4.7 (a), MCI has a few high spikes when it is necessary to merge. All other queries are very fast. MGI has much lower cost per query while MRI even lower since it avoids touching not interesting parts of the cracker column. For a lower selectivity (Figure 4.7 (b)) the relative picture is the same. MRI and MGI have a few higher peaks than before but still a lot cheaper than MCI. For the HFLV scenario, our observations are quite similar. MRI again outperforms the rest of the algorithms with a very low cost per query. In the HFLV scenario, all algorithms have quite dense peaks. This is reasonable, because by having updates more often, we also have to merge more often, and thus we have fewer tuples to merge each time. In addition, MCI has lower peaks compared to the previous scenario, but still much higher than MRI.

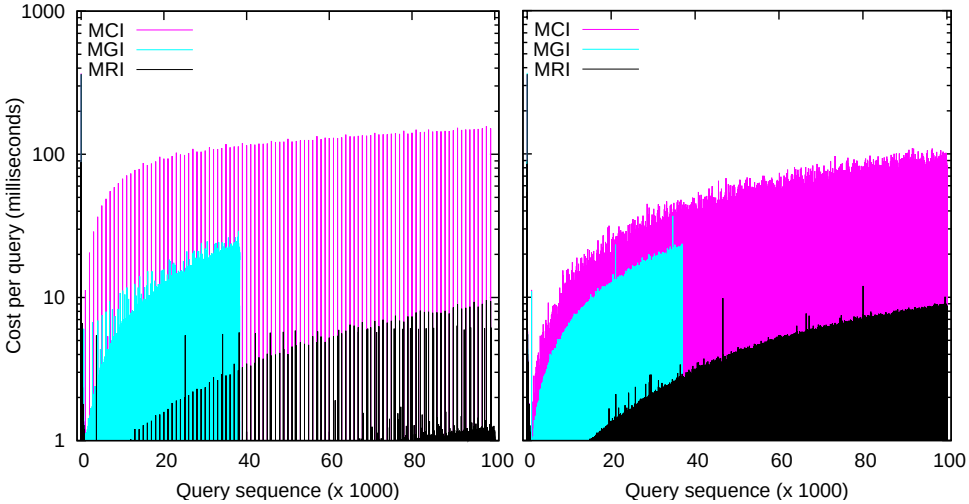
4.6.4 Long Query Sequences

All previous experiments were for a limited query sequence of 10^4 queries interleaved with updates. Here, we test for sequences of 10^5 queries. As before, we test with a column of 10^7 tuples, while the queries request random ranges that span over 10^4 values. Figure 4.8 shows the results. Compared to our previous experiments, the relative performance is not affected (i.e., MRI maintains its advantages), which demonstrates the algorithmic stability. All algorithms slightly increase their average cost per query until they stabilize after a few thousand queries. However, especially for MRI, the cost is significantly smaller than that of an AVL-tree index or the scan-select operator. The reason for observing this increase, is that with each query the cracker column is physically reorganized and split to more and more pieces. In general, the more pieces in a cracker column, the more expensive a merge operation becomes, because more tuples need to be moved around.

In order to get the very last bit of performance, our future work plans include research in allowing a cracker column/index to automatically decide to stop splitting the cracker column into smaller pieces or decide to merge existing pieces together so that the number of pieces in a cracker column can be a controlled parameter.

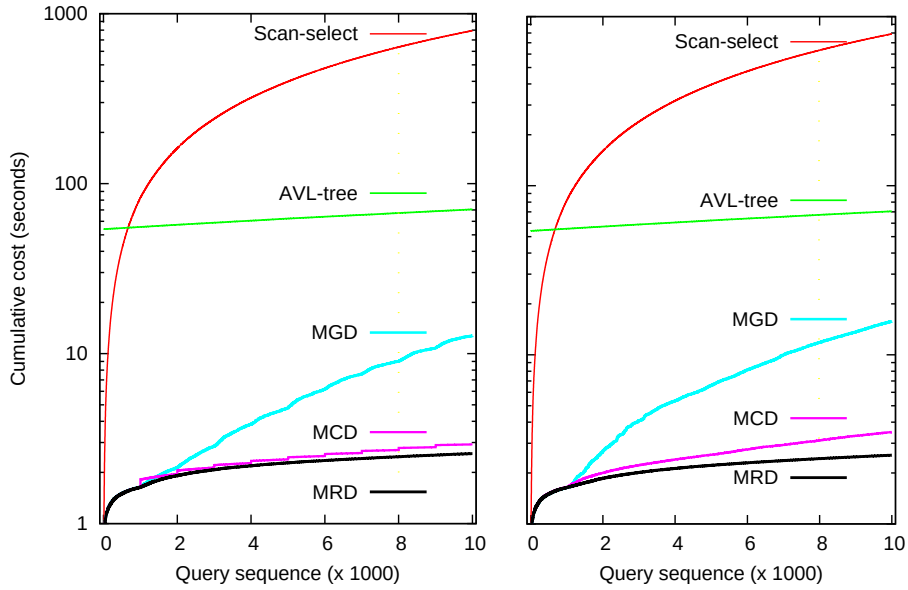


(a) LFHV (b) HFLV



(c) LFHV (d) HFLV

Figure 4.8: Effect of long query sequences



(a) LFHV scenario (b) HFLV scenario

Figure 4.9: Cumulative cost for deletes

4.6.5 Performance under Deletions

Switching our experiment focus to deletions produces similar results. The relative performance of the algorithms remains the same. For example, on a cracker column of 10^7 tuples, we fire 10^4 range queries that request random ranges of size 10^4 values. We test both the LFHV scenario and the HFLV scenario.

In Figure 4.9, we show the cumulative cost and compare it against the MonetDB scan-select that always scans a column and an AVL-tree index. The AVL-tree uses lazy deletes, i.e., spot the appropriate node and mark it as deleted so that future queries can ignore it. As with insertions, all cracker update algorithms are superior to the AVL-tree index and the scan-select. Figure 4.10 shows the cost per query (for the LFHV case), where we observe the same pattern we saw for insertions with the ripple version, the MRD algorithm, outperforming all others. The same stands for the rest of the experiments we did for deletions to see the effect of selectivity, the effect of the size of the query sequence and so

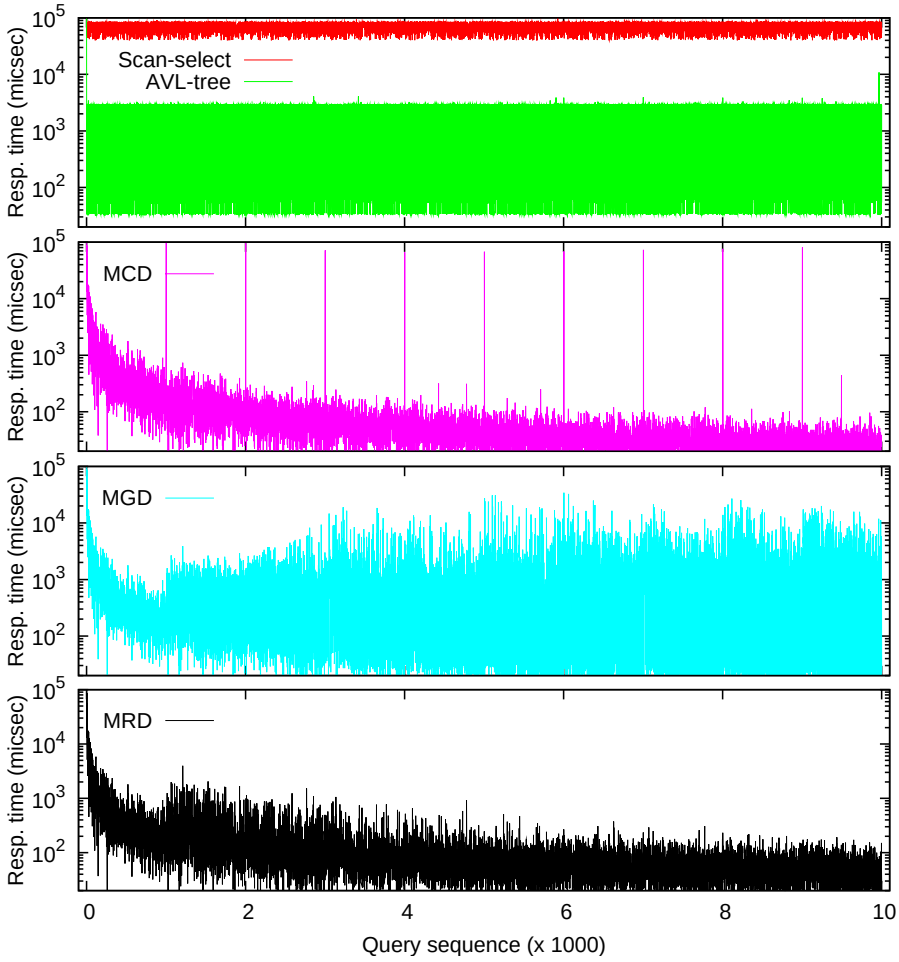


Figure 4.10: Cost per query for deletes

on. Due to space restrictions we omit these results.

An interesting difference between insertions and deletions, is that the latter requires finding the actual position for a pending deleted tuple. As we described in Section 4.4, this is done with a join operation between the respective parts of the cracker column and the pending deletions column. This is more expensive

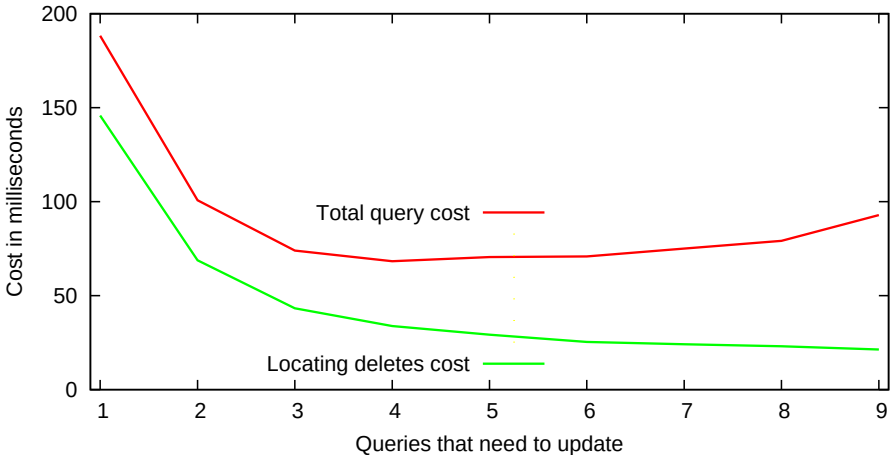


Figure 4.11: Cost to locate deletes for MCD

when the cracker pieces are large. For this reason the pattern shown graphically in Figure 4.11 is relevant. It shows only the queries that do an update for MCD in our previous experiment. We depict the total cost for each query and the cost to locate the deletes removed from the cracker column. Observe that initially, e.g., for the first query that is forced to update, the total cost is mainly due to the cost of locating tuples to be deleted. The rest of the merge process is quite cheap, since with fewer pieces in the cracker column, fewer tuples need to be moved. The next query that starts an update has a much lower total cost. It can locate deletes much faster due to having smaller pieces in the cracker column (around 10^3 queries have cracked the column in between). For the remaining update queries, the cost to locate deletes is continuously becoming smaller due to the cracker pieces becoming smaller. The total cost remains quite stable, because by having smaller pieces we also need to move more tuples while removing deletes. This pattern exists in the other algorithms, too, e.g., observe MRD in Figure 4.10. After the first thousand queries, when the first update happens, the cost per query is higher compared to that of future queries that handle smaller pieces in the cracker column.

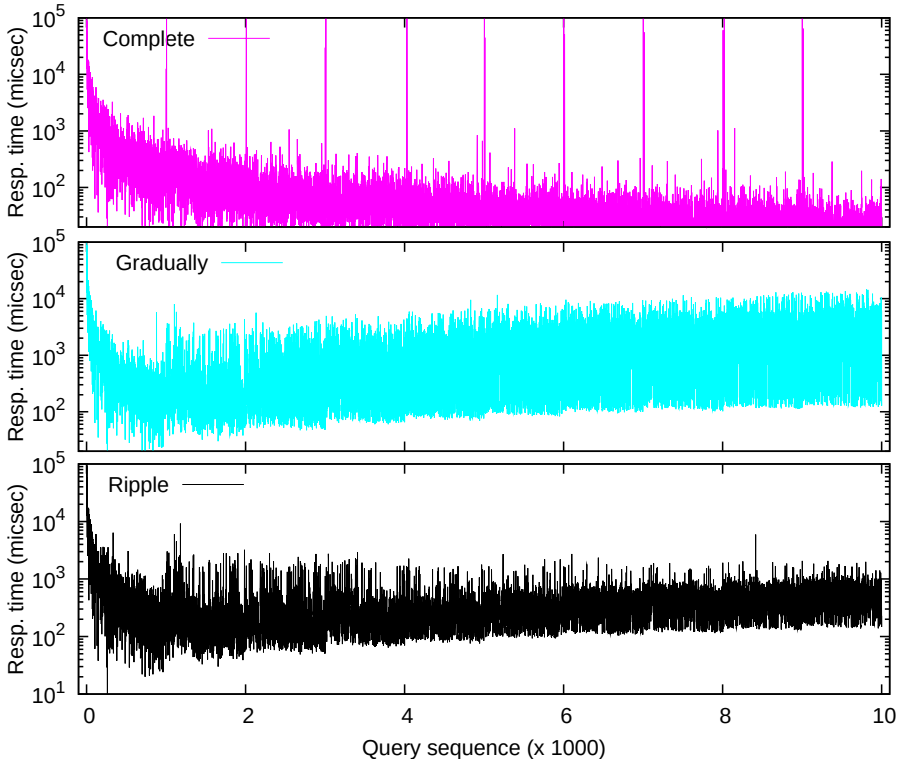


Figure 4.12: Cost per query for updates

4.6.6 Full Updates Performance

By now it should be clear that updates do not produce any surprises. The same patterns emerge, i.e., the combination of the ripple algorithms is the one that outperforms all others having the lowest and most stable cost per query along with a low total cost. Due to space restrictions (and similarity of results) we show only the cost per query for the merge-like algorithms. As before, the experiments are based on a column of 10^7 tuples, where we fire 10^4 range queries that request random ranges of size 10^4 values. A thousand updates arrive every thousand queries.

The results are shown in Figure 4.12. The only difference is that queries that need to consume both pending insertions and pending deletions cost slightly

more. For example, the combination of the gradual algorithms and the combination of the ripple algorithms never drop below 100 microseconds (as more queries arrive), which was often the case in the previous experiments. However, the relative performance is the same and still significantly lower than that of an AVL-tree or the scan-select, especially for the ripple case.

4.7 Summary

Just-enough and just-in-time are the ingredients in cracked databases. The physical store is extended with an efficient navigational index as a side product of running query sequences. It removes the human from the database index administration loop and relies on self-tuning by adaptation.

In this chapter, we extended the approach towards volatile databases. Several novel algorithms are presented to deal with database updates using the cracking philosophy. The algorithms were added to an existing open-source database kernel and a broad range experimental analysis, including a comparison with a competitive index scheme, clearly demonstrates the viability of cracking in column-stores.

With these promising results the road for many more discoveries of self-* database techniques lies wide open. The following chapters show how we can exploit cracking for efficient tuple reconstruction and arbitrary join processing in a column-store.

Chapter 5

Sideways Cracking*

5.1 Introduction

The previous chapters have mainly focused on single-attribute queries. Here, we make the next step towards a fully functional cracking DBMS; we study *complex multi-attribute queries*.

Database systems are extremely complex pieces of software; the slightest change in a core functionality may have significant effects in multiple areas. We show that the physical reorganization step in selection cracking leads to random access patterns during tuple reconstruction, resulting in a significant performance decrease for multi-attribute queries.

In fact tuple reconstruction itself is probably the most important operator in a column-store. It is in practice a join operation and typically reflects a dominant cost portion (in terms of total query response time) and needs to be performed multiple times within a single query plan. This way, efficient tuple reconstruction is a key component in a column-store.

In this chapter, we reconsider the initial and minimal cracking architecture with all this in mind. Our goal is to maintain the benefits and behavior we have seen so far and to be able exploit the cracking philosophy in complex queries transferring the benefits of cracking throughout the database kernel. This chapter achieves that by making tuple reconstruction an integral part of database cracking.

*The material in this chapter has been the basis for the ACM SIGMOD09 paper "Self-organizating Tuple Reconstruction In Coumn-stores" (Idreos et al., 2009).

The Ultimate Access Pattern

Tuple reconstruction is a crucial aspect of a column-store design. We can largely distinguish between early and late tuple reconstruction with the latter giving a large number of benefits by allowing to process data one column at a time in a vector-like way. Chapter 2 gives a more detailed introductory discussion on these issues. Access patterns is the key for efficient tuple reconstruction and column-stores try to exploit the positional alignment of columns and to ensure that whenever possible tuple reconstruction is performed with sequential access patterns.

The ultimate access pattern is to have multiple copies for each relation R, such that each copy is presorted on an other attribute in R. All tuple reconstructions of R attributes initiated by a restriction on an attribute A can be performed using the copy that is sorted on A. This way, the tuple reconstruction does not only exploit sequential access, but also significantly benefits from focused access to only a small consecutive area in the base column (as defined by the restriction on A) rather than scattered access to the whole column.

However, such a direction requires the ability to *predict* the workload and the luxury of *idle time* to prepare the physical design. In addition, up to date there is no efficient way to maintain multiple sorted copies under updates in a column-store; thus it requires read-only or infrequently updated environments.

5.1.1 Contributions

In this chapter, we propose a self-organizing direction based on cracking. It achieves tuple reconstruction performance similar to using presorted data, but comes without the hefty initial price tag of presorting itself. Instead, it handles dynamic unpredictable workloads with frequent updates and with no need for idle time.

The initial cracking architecture, selection cracking, seen in previous chapters, is not sufficient since as we will show the one column at a time reorganization leads to random access patterns during tuple reconstruction. The new architecture pushes cracking even deeper into the kernel design.

We introduce a novel design, *partial sideways* cracking, that provides a self-organizing behavior for both selections and tuple reconstructions. It gracefully handles any kind of complex multi-attribute query. It uses auxiliary self-organizing data structures to materialize mappings between pairs of attributes used together in queries for tuple reconstruction. Based on the workload, these *cracker maps* are continuously kept aligned by being physically reorganized,

while processing queries, allowing the DBMS to handle tuple reconstruction using cache-friendly access patterns.

To enhance performance and adaptability, in particular in environments with storage restrictions, cracker maps are implemented as dynamic collections of physically separate chunks. This enables flexible storage management by adaptively maintaining only those chunks of a map that are required to process incoming queries. Chunks adapt individually to the query workload. Each chunk of a map is separately reorganized, dropped if extra storage space is needed, or recreated (entirely or in parts) if necessary.

We implemented partial sideways cracking on top of MonetDB. This chapter describes in detail the new algorithms, operators and query plans. We present an extensive experimental analysis using both synthetic workloads and the TPC-H benchmark. We clearly show that partial sideways cracking brings a self-organizing behavior and significant benefits especially for multi-attribute queries with complex non order-preserving operators, even in the presence of random workloads, storage restrictions and updates.

5.1.2 Outline

The remainder of this chapter is organized as follows. Section 5.2 demonstrates the tuple reconstruction problem. Then, to enhance readability and fully cover the research space, we present partial sideways cracking in two steps. Focusing on the tuple reconstruction problem and neglecting storage restrictions at first, Section 5.3 introduces the basic sideways cracking technique using fully materialized maps, accompanied with an extensive experimental analysis. Then, Section 5.4 extends the basic approach with flexible storage management using partial maps. Detailed experiments demonstrate the significant benefits over the initial full materialization approach. Then, Section 5.5 presents the benefits of sideways cracking with the TPC-H benchmark. Finally, Section 5.6 concludes the chapter.

5.2 The Tuple Reconstruction Problem

In this section, we show that tuple reconstruction is a dominant cost factor in column-stores, and that it becomes a challenge when selection cracking is applied in multi-attribute queries. This phenomenon, along with the key components towards a solution, is illustrated using a series of simple experiments before we continue with sideways cracking in subsequent sections.

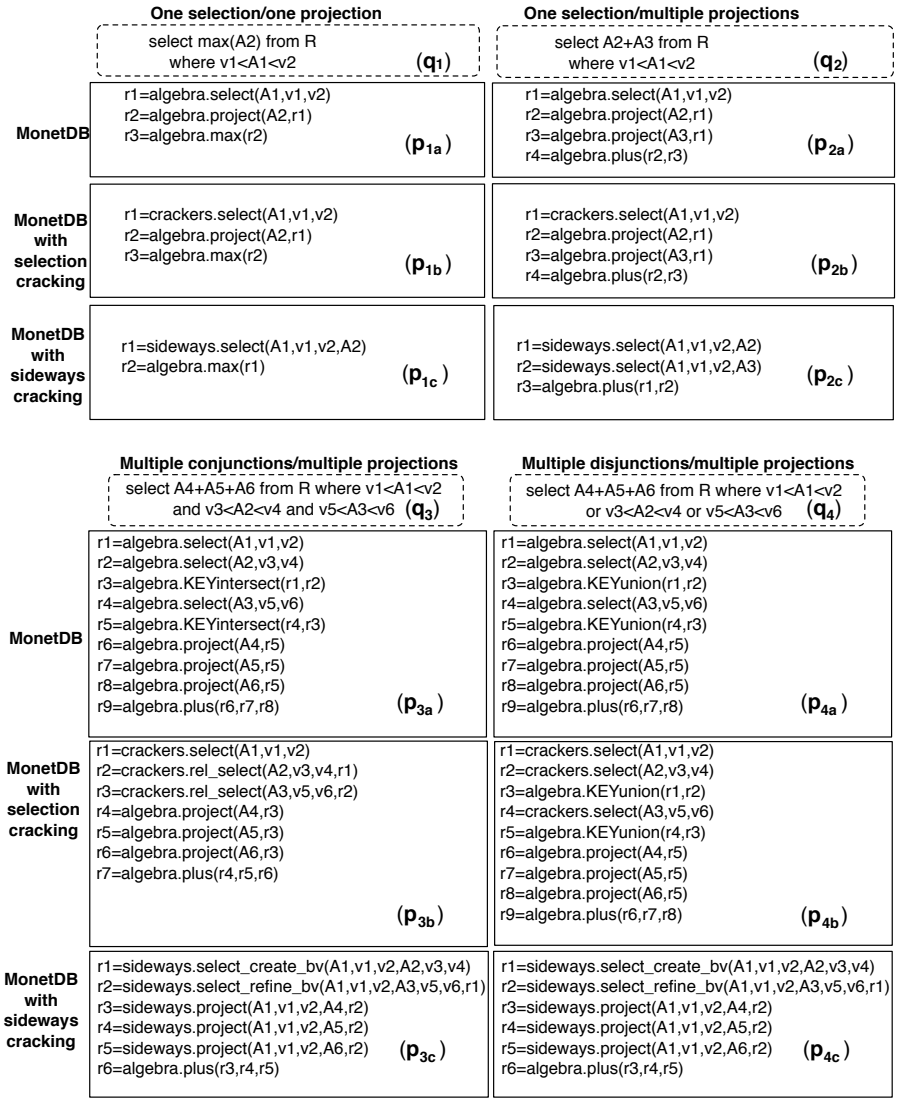


Figure 5.1: Sample Queries and their MonetDB plans with and without cracking

5.2.1 Example Queries

Throughout this chapter we will use four example queries to demonstrate how query plans are affected by our architectural decisions in database cracking. Figure 5.1 illustrates how these four relational queries ($q_1 - q_4$) are translated into the respective binary-relational MonetDB plans ($p_{1a} - p_{4a}$). MonetDB whenever possible exploits the positional alignment of the columns during tuple reconstruction. The descriptions of the operators used in Figure 5.1 can be found in Section 2.10.

Figure 5.1 also shows how MonetDB with selection cracking enabled translates these queries ($p_{1b} - p_{4b}$). These plans exploit the new selection cracking operators as introduced in Chapter 3. Recall from our previous discussions that the cracking select operator returns a collection of keys/positions exactly as `algebra.select()` of normal MonetDB does. However, due to physical reorganization, cracker columns are no longer aligned with base BATs and consequently the selection results are no longer ordered according to the tuple insertion sequence. For queries with multiple selections (q_3 and q_4), we need to perform the intersection/union of individual selection results on unordered `key`-value sets. Allowing neither sequential access nor positional lookups, performance worsens significantly for tuple reconstruction. For this reason, for conjunctive queries, selection cracking uses `crackers.select` only for the first selection, introducing the `crackers.rel_select` operator for all subsequent selections. This operator performs the tasks of `algebra.select` and `algebra.KEYintersect` in one go. For each `key`-value of the previous selection's result, the next selection's attribute value is derived from its base BAT exploiting positional (though unordered) lookup, and the `key`-value is added to the result only if the attribute value satisfies the condition. With multiple selections, both `crackers.rel_select` and `algebra.project` may suffer from unordered (though positional) lookups into BATs. We will study in detail these issues here, demonstrating that we need more deep solutions as queries become more complex.

Later, as this chapter evolves, we will discuss how and why sideways cracking rewrites the queries into the new plans shown in ($p_{1c} - p_{4c}$) not only avoiding the overhead of selection cracking but also providing tuple reconstruction close to the optimal one as it would be achieved by a presorting strategy but without the restrictions that come with pre-sort.

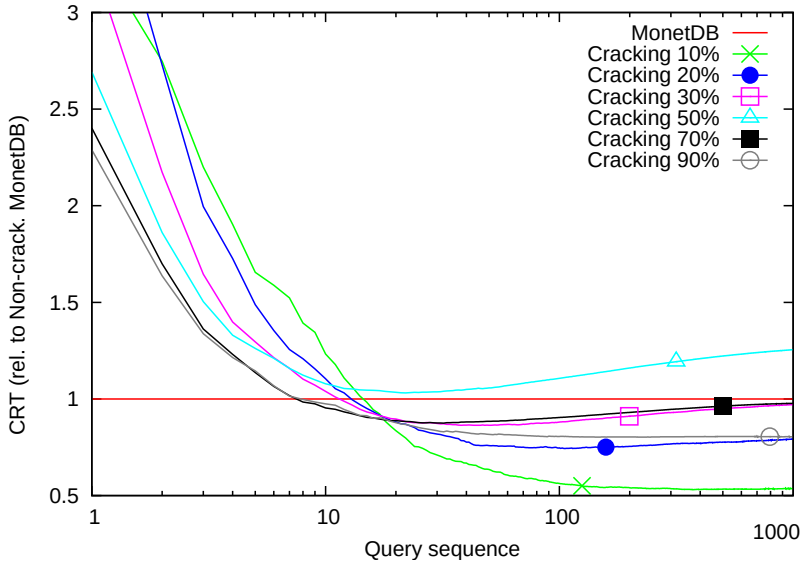


Figure 5.2: Decreasing performance due to tuple reconstruction cost (CRT: cumulative response time)

5.2.2 Experimental Analysis

We continue with a series of experiments to demonstrate that random access patterns when reconstructing tuples can decrease performance in multi-attribute queries. We compare the latest non-cracking version of MonetDB against selection cracking developed on top of MonetDB. We use a 2.4 GHz AMD Athlon 64 processor equipped with 2 GB RAM and two 250 GB 7200 rpm S-ATA hard disks configured as software-RAID-0. The operating system is Fedora Core 6 (Linux 2.6.20). All experiments are on a relational table of 8 attributes (A_1 to A_8), all containing randomly distributed integers in $[1, 10^7]$.

Exp1: Basic Performance

We run 1000 queries of type q_1 in Figure 5.1, selecting randomly located ranges. This simple query requires only one tuple reconstruction performed by a project operator. The experiment is repeated for various selectivity factors $\{10\%, \ldots, 90\%\}$. To make the differences more prominent, Figure 5.2 shows the cumulative

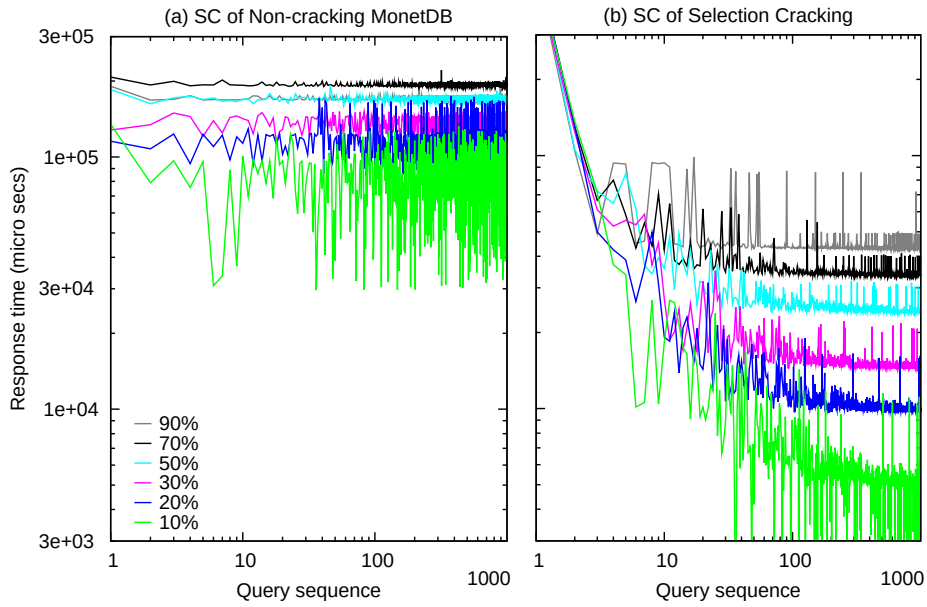


Figure 5.3: Selection cost (SC) for various selectivity factors

response time (CRT) relative to the performance of non-cracking MonetDB (per selectivity factor).

For cracking we observe a self-organizing behavior. The first query is slightly slower since the appropriate cracker column needs to be created when an attribute is used for the first time. After only a few queries, the cracking data structures are physically reorganized to an extend that significantly fewer non-qualifying tuples have to be accessed to answer queries. However, notice that as the selectivity factor, and hence the result size, grows, the selection cracking approach looses its advantage against MonetDB. The performance drop of selection cracking peaks at 50% selectivity, and then improves again.

Figures 5.3 and 5.4 help to analyze this behavior. They show separately the per query selection and tuple reconstruction costs. Any variances are due to caching effects between subsequent queries requesting overlapping ranges. For the selection costs in Figure 5.3, we see the "typical" self-organizing behavior of cracking, i.e., the more queries arrive the more cracking learns about the data

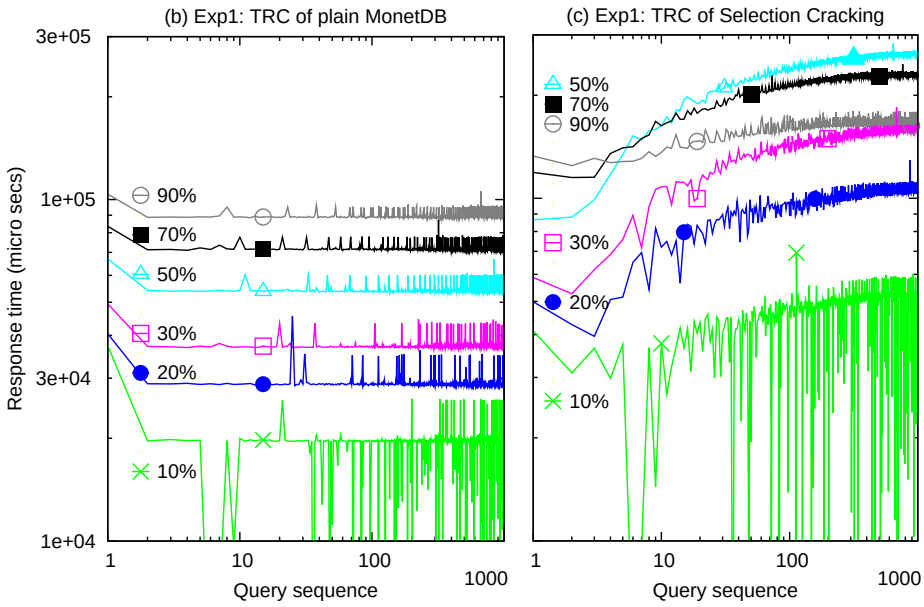


Figure 5.4: Tuple reconstruction cost (TRC) for various selectivity factors

organization and the more it can improve access patterns during selections. This way, cracking selections continuously become faster and faster over the plain scan select of normal MonetDB. This is the same behavior as we observed and discussed in Chapter 3 as well.

On the other hand, Figure 5.4 shows that the tuple reconstruction cost is significantly higher for cracking. Here, we see the exact opposite behavior; as more queries arrive, the more expensive tuple reconstruction becomes. This is a result of the access patterns used. With non-cracking MonetDB, `algebra.select` is order-preserving, hence, tuple reconstruction for the project operator is performed using in-order positional `key`-lookups into the projection attribute's base BAT. The resulting sequential access pattern is very cache-friendly ensuring that each page or cache-line is loaded at most once. On the contrary, with selection cracking, the result of `cracker.select` is no longer aligned with the base BATs. Consequently, the tuple reconstruction is performed using *randomly ordered* positional `key`-lookups into the base BAT. Lacking both spatial and temporal lo-

cality, the random access pattern causes significantly more cache-/page-misses, making tuple reconstruction more expensive. Moreover, the order of tuple-`keys` in a cracker column becomes more random, the more queries contribute to cracking, resulting in further increasing tuple reconstruction costs.

In addition, observe that the selection costs show the expected behavior, growing monotonously with increasing selectivity factors (i.e., result size). The tuple reconstruction costs of selection cracking, however, increase only up to 50% selectivity, and then decrease, again. The increase is obvious from the growing result sizes. The decrease is the result of two characteristics that effectively make the tuple reconstruction more cache-friendly. First with selectivity factors larger than 50%, the cracker column is less fragmented, as queries always request large contiguous chunks. Consequently, larger chunks in the cracker column are still "almost aligned" with the base BAT. Second, with larger result sizes, the result tuples form a more dense subset of the base BAT, increasing the probability that subsequently accessed tuples (though in random order) "happen" to be on the same page/cache-line. The increased locality in the resulting access pattern reduces the number of page-/cache-misses and thus reduces the tuple reconstruction costs.

This experiment demonstrates that even for queries requiring only one tuple reconstruction, cracking can loose its performance advantage for certain selectivity factors. Next we study queries with multiple tuple reconstructions.

Exp2: Multiple Tuple Reconstructions

The next experiment demonstrates the behavior in query plans with one selection as before, but now with multiple projection operations (q_2 in Fig. 5.1). We test with queries that use 2 to 8 attributes in the select clause to perform some simple aggregations, e.g., $\max(A_1)$, $\max(A_2)$, $\cdots$. For each case we run 1000 queries that request a random range of 20% of the tuples. With such selectivity, selection cracking outperformed non-cracking MonetDB on simple queries in the previous experiment.

Figure 5.5 shows the results. Naturally, with more tuple reconstruction operations involved, we observe a growing dominance of the expensive random-access tuple reconstruction, eliminating the benefits of selection cracking over the basic non-cracking approach for queries that access more than two attributes. The overall effect is that non-cracking MonetDB always outperforms cracking when more than two attributes participate in the query.

Finally, we observe a correlation between the number of projection operators and the first query cost in the sideways cracking case. It reflects the initial cost

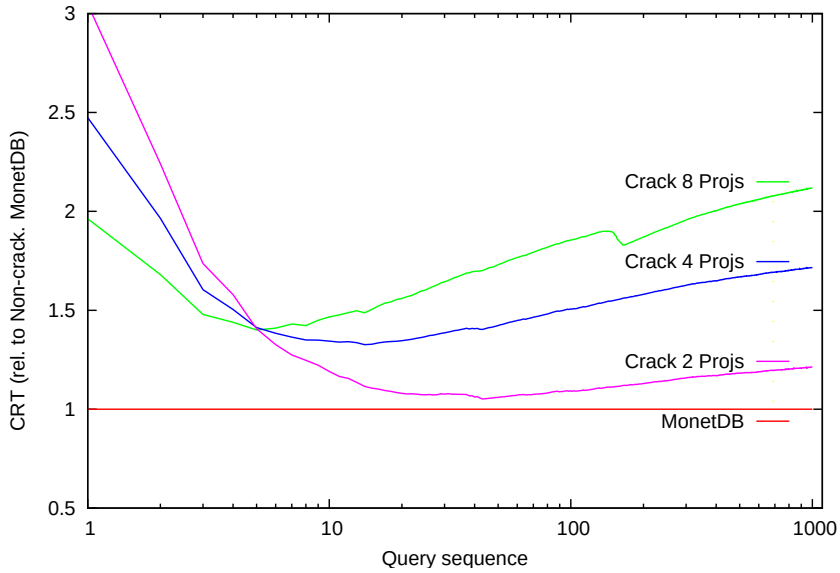


Figure 5.5: Multiple tuple reconstructions (CRT: cumulative response time)

to create the necessary cracker maps. However, notice that typically this is not a cost incurred by a single query since multiple queries are run against a database system. A query has to create a cracker map only if it does not already exist. Different selectivity factors lead to similar results.

Exp3: Reordering intermediate results

A natural direction to improve selection cracking tuple reconstructions is to invest in reordering unordered intermediate results aiming at a less random access pattern. For example, we can sort based on keys the intermediate result of a crack select operator. Sorting ensures sequential access during reconstruction but . the preparatory sorting step is expensive.

As an alternative, we can use a cache-friendly radix-clustering algorithm — originally proposed in (Boncz et al., 1999; Manegold et al., 2002; Manegold et al., 2004) to support efficient cache-aware join processing — to create sub-cache size blocks that restrict random access during reconstruction to the cache. This achieves similar reconstruction performance as purely sequential access at

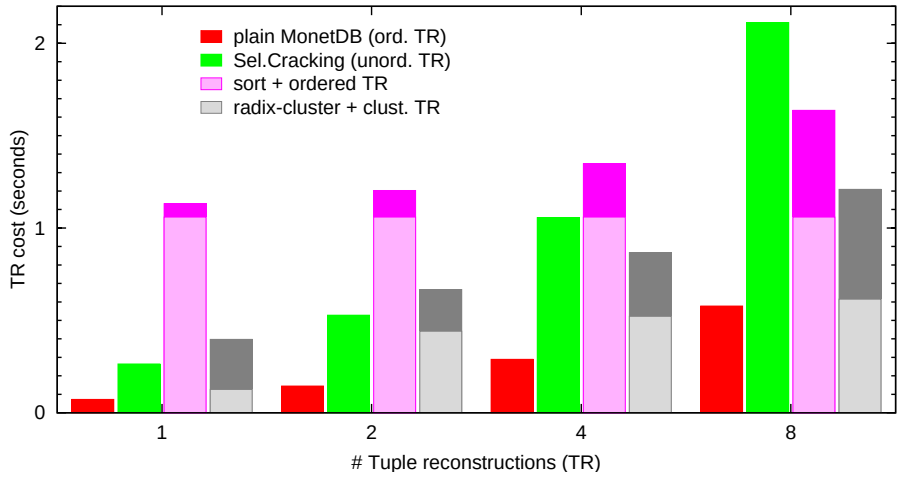


Figure 5.6: Reordering intermediate results

a lower investment than sorting.

Figure 5.6 compares the tuple reconstruction cost of plain MonetDB (ordered selection result) and selection cracking (unordered selection result) with alternatives that sort or radix-cluster the selection result. Since for a given query plan, we know up-front the number of reconstructions that will benefit, we appropriately tune the degree of radix-clustering, or in other words, the investment cost. This way, these results reflect the best possible performance for the radix case. The investment in clustering (sorting) pays off with 4 (8) or more projections.

In this way, reordering intermediate results pays off when multiple projections share a single intermediate result. However, reordering is not beneficial with only a few projections or when we have multiple selections, where individual intermediate results prohibit the sharing of reordering investments.

Exp4 & Exp5: Multiple Selections

The next experiment studies queries with multiple tuple reconstructions due to multiple selections (as q_3 & q_4 in Fig. 5.1), but no projections, i.e., the select clause is a simple $count(*)$. Thus, tuple reconstruction is performed when trying to combine the various selection results. We test with queries that use 4

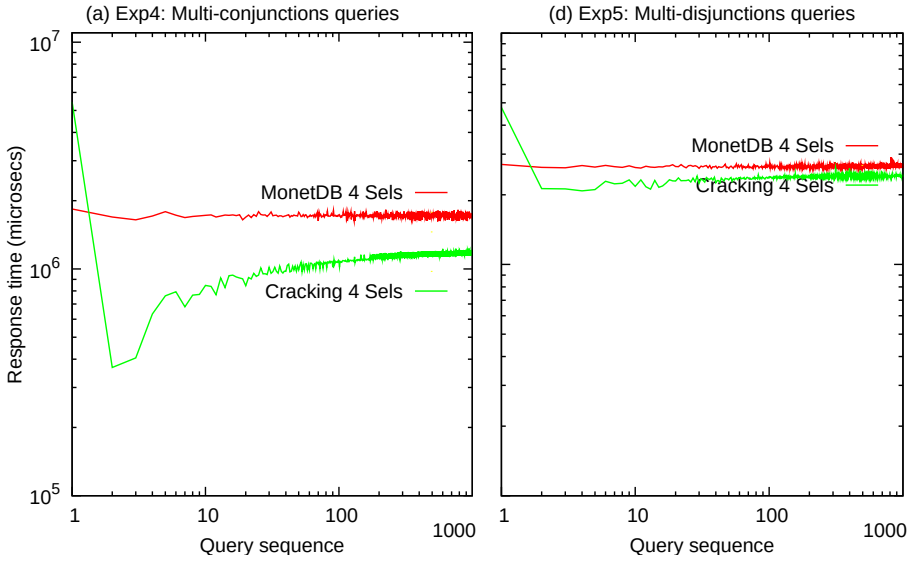


Figure 5.7: Decreasing performance due to tuple reconstruction cost in multi-selection queries

selections. For conjunctive queries, the first selection is 50% selective and each subsequent one is 5% more selective. For disjunctive queries, every selection gets 10% of the tuples. We run 1000 queries. Figures 5.7(a) and 5.7(b) show the per query cost for conjunctive and disjunctive queries, respectively.

Again the multiple tuple reconstruction operations significantly affect the overall cost in such a way that selection cracking cannot materialize a benefit in terms of total query processing time. Even with the use of the `crackers.rel_select` operator, in the case of conjunctive queries, selection cracking looses significant ground due to the random memory access patterns. Experiments with more conjunctions/disjunctions naturally lead to even worse performance since more tuple reconstructions need to be performed.

Summary

The lesson learned from these experiments is that although selection cracking brings a significant improvement in the selection operation over a single

attribute, when it comes to multi-attribute queries, it cannot materialize this benefit due to the random access patterns when reconstructing tuples. The order of tuples in the columns plays a crucial role and having base columns and intermediate results aligned is the key for efficient cache-conscious tuple reconstructions. In the rest of this chapter, we present our solution, sideways cracking, that always uses cache-friendly sequential access patterns to reconstruct tuples.

5.3 Sideways Cracking

In this section, we introduce the basic *sideways cracking* technique using fully materialized maps. To motivate and illustrate the effect of our choices, we build up the architecture starting with simple examples before continuing with more flexible and complex ones. The section closes with an experimental analysis of sideways cracking with full maps against the selection cracking and non-cracking approaches. The addition of adaptive storage management through partial sideways cracking is discussed and evaluated in Section 5.4. Section 5.5 shows the benefits on the TPC-H benchmark.

5.3.1 Basic Definitions

We define a *cracker map* M_{AB} as a two-column table over two attributes A and B of a relation R. Values of A are stored in the left column, while values of B are stored in the right column, called *head* and *tail*, respectively. Values of A and B in the same position of M_{AB} belong to the same relational tuple. A cracker map is represented in MonetDB as a BAT. Given a relational table R with k attributes, up to $k-1$ cracker maps can potentially exist for each attribute of R, representing all ordered combinations of two attributes. All maps that have been created using A as head are collected in the *map set* S_A of $R.A$. Maps are created *on demand*, only. For example, when a query q needs access to attribute B based on a restriction on attribute A (e.g., as q_1 in Figure 5.1) and M_{AB} does not exist, then q will create it by performing a scan over base columns A and B. For example, query q_1 is a typical example that can trigger the creation and/or the usage of a cracker map. In this case q_1 triggers the creation of M_{AB}.

For each cracker map M_{AB}, there is a *cracker index* (AVL-tree) that maintains information about how A values are distributed over M_{AB}.

Once a map M_{AB} is created by a query q, it is used to evaluate q and it stays alive to speed up data access in future queries that need to access B based on

A. Each such query triggers physical reorganization (cracking) of M_{AB} based on the restriction applied to A. Reorganization happens in such a way, that all tuples with values of A that qualify the restriction, are in a *contiguous* area in M_{AB}. Cracker maps are inspired by the cracker columns of (Idreos et al., 2007a). The difference is that a cracker column holds attribute values and their keys while cracker maps hold related values of two attributes. In principle, a cracker column C_A can be seen as a cracker map $M_{A\texttt{key}}$. We use the two algorithms of (Idreos et al., 2007a) to physically reorganize maps by splitting a piece of a map into two or three new pieces.

We introduce `sideways.select(A,v1,v2,B)` as a new selection operator that returns tuples of attribute B of relation R based on a predicate on attribute A of R as follows:

(1) If there is no cracker map M_{AB}, then create one.
(2) Search the index of M_{AB} to find the contiguous area w of the pieces related to the restriction σ on A.
    If σ does not match existing piece boundaries,
    (3) Physically reorganize w to move false hits out of the contiguous area of qualifying tuples.
    (4) Update the cracker index of M_{AB} accordingly.
(5) Return a non-materialized view of the tail of w.

With sideways cracking we get plan p_{1c} for query q_1 in Figure 5.1. Notice the absence of a project operator compared to p_{1a} and p_{1b}. Hence, the result of `sideways.select` is already the result of the whole query, requiring no tuple reconstruction.

Example. Before extending sideways cracking to efficiently support more complicated queries containing multi-attribute selections and projections, we first give a simple example. Assume a relation $R(A, B)$ shown in Figure 5.8. Initially, there are no maps available. The first query requests values of B where a restriction on A holds. The system creates map M_{AB} and cracks it into three pieces based on the selection predicate. All tuples with qualifying values of A are clustered in the middle piece. The first piece contains tuples with values lower than that of the result, while the third piece contains tuples with values higher than the result. Via cracking the qualifying B values are already clustered together *aligned* with the qualifying A values. Thus, no explicit join-like operation is needed for tuple reconstruction; the tail column of the middle piece forms the query's result. For such simple cases, sideways cracking benefits from the fact that the used map contains both the selection and the projection attribute.

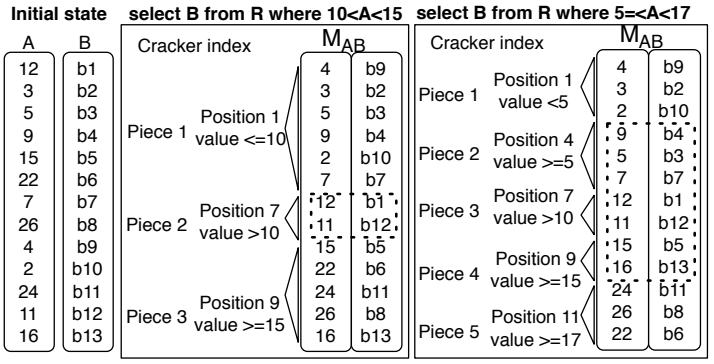


Figure 5.8: A simple example

Observe that the cracker index contains now information to guide future queries. Then, a similar second query arrives. From the index, we derive that (1) the entire middle piece belongs to the result, and hence, (2) only Pieces 1 and 3 must be analyzed and further cracked. Each of them is further partitioned into two new pieces using single-sided cracking. Then, the result for the second query is again in a contiguous area; this time consisting of a concatenation of Pieces 2, 3 and 4. As more queries are being processed, the system "learns" — purely based on incoming queries — more about how data is clustered, and hence, can reduce data access. A cracker map continuously adapts to the expressed user interest. Only parts of interest to the users have been analyzed, the rest remains an uncharted area and will be explored only when user focus shifts. No external human administration is needed since the system adapts based on user requests.

5.3.2 Multi-projection Queries

Let us now discuss queries with multiple tuple reconstruction operations. We start with queries over a single selection and multiple projections. Then, Section 5.3.3 discusses queries with multiple selections.

The Problem: Non-aligned Cracker Maps

Naturally, a single-selection query q that projects n attributes requires n maps, one for each attribute to be projected. These maps M_{Ax} belong to the same

map set S_A. However, a naive use of the maps can lead to incorrect query results. Consider the example depicted in the upper part of Figure 5.9. The first query triggers the creation of M_{AB} and it physically reorganizes it based on $A<3$. Similarly, the second query triggers the creation of M_{AC} and cracks it according to $A<5$. Then, the third query needs both M_{AB} and M_{AC}. Refining both maps according to $A < 4$ of the third query creates correct consecutive results for each map individually. However, since these maps had previously been cracked independently using different restrictions on A, the result tuples are not positionally aligned anymore, prohibiting efficient positional result tuple reconstruction. Maintaining the tuple identity explicitly by adding a `key` column to the maps is not an efficient solution, either. It increases the storage requirements and allows only expensive join-like tuple reconstruction requiring random access due to non-aligned maps.

The Solution: Adaptive Alignment

To overcome this problem, we extend the `sideways.select` operator with an *alignment step* that adaptively and on demand restores the alignment of all maps used in a query plan. The basic idea is to apply *all* physical reorganizations, due to selections on an attribute A, in the *same order* to all maps in S_A. Due to the deterministic behavior of the cracking algorithms (Idreos et al., 2007a), this approach ensures alignment of the respective maps.

Obviously, in an unpredictable environment with no idle system time, we want to invest in this extra work only if it pays-back, i.e., only once a map is required. In fact, performing alignment on-line is not an option. On-line alignment would mean that *every* time we crack a map, we also forward this cracking to the rest of the maps in its set. This is prohibitive for several reasons. First, in order to be able to align all maps in one go we need to actually materialize and maintain all possible maps of a set, even the ones that the actual workload does not require. Most importantly every query would have to touch all maps of a set, i.e., *all* attributes of the given relation. This immediately overshadows the benefit of using a column-store in touching only the relevant attributes every time. The overhead of having adaptive alignment is that each map M_{Ax} in a set S_A needs to materialize the head attribute A so that M_{Ax} can be cracked independently. We will remove this restriction with partial sideways cracking in the next section.

To achieve adaptive alignment, we introduce a *cracker tape* T_A for each set S_A, which logs (in order of their occurrence) all selections on attribute A that trigger cracking of any map in S_A. Each map M_{Ax} is equipped with a *cursor*

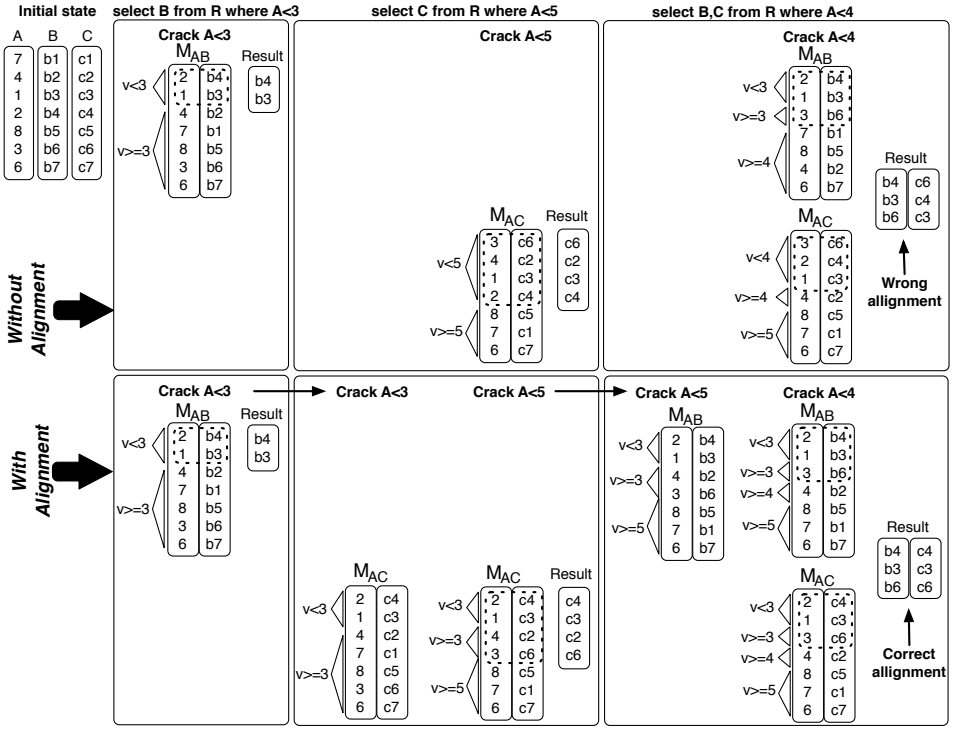


Figure 5.9: Multiple tuple reconstructions in multi-projection queries

pointing to the entry in T_A that represents the last crack on M_{Ax}. Given a tape T_A, a map M_{Ax} is aligned (synchronized) by successively forwarding its cursor towards the end of T_A and incrementally cracking M_{Ax} according to all selections it passes on its way. All maps whose cursors point to the same position in T_A, are physically aligned.

To ensure that alignment is performed on demand only, we integrate it into query processing. When a query q needs a map M, then and only then, q aligns M. We further extend the `sideways.select(A,v1,v2,B)` operator with three new steps that maintain and use the cracker tapes as follows:

(1) If there is no T_A, then create an empty one.
(2) If there is no cracker map M_{AB}, then create one.
(3) Align M_{AB} using T_A.
(4) Search the index of M_{AB} to find the contiguous area w of the pieces related to the restriction σ on A.
If σ does not match existing piece boundaries,
(5) Physically reorganize w to move false hits out of the contiguous area of qualifying tuples.
(6) Update the cracker index of M_{AB} accordingly.
(7) Append predicate $v_1 < A < v_2$ to T_A.
(8) Return a non-materialized view of the tail of w.

For a query with one selection and k projections, the query plan contains k `sideways.select` operators, one for each projection attribute. For example, assume a query that selects on A and projects B and C. Then, one `sideways.select` operator will operate over M_{AB} and another over M_{AC}. With the maps aligned and holding the projection attributes in the tails, the result is readily available.

The bottom part of Figure 5.9 demonstrates how queries are evaluated using aligned maps yielding the correctly aligned result. The first query works as before, but the second one is evaluated in a different way. After creating M_{AC}, it first applies the reorganizations that have previously been applied to M_{AB} by the first query. Thus, M_{AB} and M_{AC} are now physically aligned. Though not required for the correctness of the second query, this step is crucial for the correctness of future queries. Then, M_{AC} is physically reorganized based on $A < 5$ to actually evaluate the second query. For the third query, we first need to align M_{AB} with M_{AC} by applying the cracking used for the second query. Then, we can crack both M_{AB} and M_{AC} based on the restriction of the third query yielding the correctly aligned result.

In this way, for both single and multiple-projections queries that select over a single attribute, sideways cracking leads to very simple query plans where only crack operations are used. For example, compare the above plan against p_{2a} and p_{2b} in Figure 5.1. In sideways cracking, there is no need to spent I/O and CPU cycles in fetching the projected attributes. We show in our experimental analysis that such query plans bring a significant improvement in response time compared to traditional approaches.

Sideways cracking performs tuple reconstruction by efficiently maintaining aligned maps via cracking instead of using (random-access) position-based joins.

The alignment step follows the self-organizing nature of a cracking DBMS.

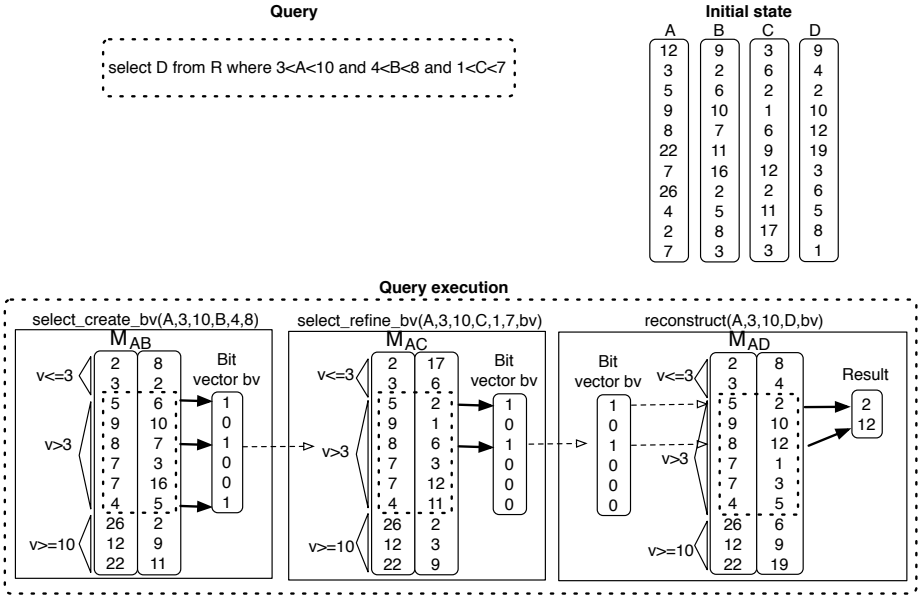


Figure 5.10: Multiple tuple reconstructions in multi-selection queries

Aligning a map M becomes less expensive the more queries use M, as incremental cracking successively reduces the size of pieces and hence the data that needs to be accessed. Moreover, the more frequently M is used, the fewer alignment steps are required per query to bring it up-to-date. Unused maps do not produce any processing costs.

5.3.3 Multi-selection Queries

In the previous sections, we showed how to evaluate queries that restrict only one attribute in the where clause. The final step is to generalize sideways cracking for queries that select over multiple attributes. One approach is to create *wider* maps that include multiple attributes in different orderings. However, the many combinations, orderings and predicates in queries lead to huge storage and maintenance requirements. Furthermore, wider maps are not compatible with the cracker algorithms of (Idreos et al., 2007a) and the update techniques of (Idreos et al., 2007b). Instead, we propose a solution that exploits aligned

two-column maps in a way that enables efficient cache-friendly operations.

The Problem: Non-aligned Map Sets

Let us first consider conjunctive queries, e.g., the query of Figure 5.10. A query plan could use maps M_{AD}, M_{BD} and M_{CD}. These maps belong to *different* sets S_A, S_B and S_C, respectively. However, the alignment techniques presented before apply only to multiple maps within the *same* set. Keeping maps of different sets aligned is not possible at all, as each attribute requires/determines its own individual order for its maps. Thus, using the above map sets for the example query inherently yields non-aligned individual selection results requiring expensive operations for subsequent tuple reconstructions.

The Solution: Use a Single Aligned Set

The challenge for multi-selections is to find a solution that uses maps of only *one single* set, and thus can exploit their alignment. We postpone the discussion about how to choose this one set till later in this section. To sketch our approach using the query of Figure 5.10 as example, we arbitrarily assume that set S_A is chosen, i.e., we use maps M_{AB}, M_{AC} and M_{AD}.

Each map is first aligned to the most recent crack operation on A and only then it is cracked given the current predicate on A. As discussed above, the other attributes cannot contribute to further cracking as this would destroy the alignment of maps. However, Given the conjunctive predicate, we know that we just created contiguous areas w_B, w_C and w_D aligned across the involved maps that contain all result candidates. These areas are aligned since all maps were *first* aligned and *then* cracked based on the *same* predicate. Thus, all areas have also the same size k. To filter out the "false candidates" that fulfill the predicate on A, but not all other predicates, we use *bit vector* processing (X100 (Boncz et al., 2005) and the study of (Abadi et al., 2007) also exploit bit-vectors for filtering multiple predicates). Using a single bit vector of size k, if a tuple fulfills the predicate on B, the respective bit is set, otherwise cleared. Successively iterating over the aligned result areas in the remaining maps (w_C in our example), the bits of tuples that do not fulfill the respective predicate are cleared. Finally, the bit vector indicates which w_D tuples form the result. An example in Figure 5.10 illustrates the details using the following three new operators.

```
sideways.select_create_bv(A,v1,v2,B,v3,v4)
```

(1-7) Equal to `sideways.select` in Section 5.3.2.

(8) Create and return bit vector bv for w with $v_3 < B < v_4$.

`sideways.select_refine_bv(A,v1,v2,B,v3,v4,bv)`

(1-7) Equal to `sideways.select` in Section 5.3.2.

(8) Refine bit vector bv with $v_3 < B < v_4$ and return bv.

`sideways.reconstruct(A,v1,v2,B,bv)`

(1-7) Equal to `sideways.select` in Section 5.3.2.

(8) Create and return a result that contains the tail value of all tuples from w in M_{AB} whose bit is set in bv.

Given the alignment of the maps and the bit vector, only positional lookups and sequential access patterns are involved. In addition, by clustering and aligning relevant data via cracking, the system needs to analyze only a small portion of the involved columns (equal to the size of the bit vector) for selections and tuple reconstructions.

Map Set Choice: Self-organizing Histograms

The remaining issue is to determine the appropriate map set. Our approach is based on the core of the "cracking philosophy", i.e., in an unpredictable environment with no idle system time, always perform the minimum investment. Do just enough operations to boost the current query. Do not invest in the future unless the benefit is known, or there is the luxury of time and resources to do so. In this way, for a query q, a set S_A is chosen such that the restriction on A is the most *selective* in q, Its rationale aligns with pushing selections down an operator tree, close to the base tables. The net effect is a minimal bit vector size, which improves the overall performance by loading and analyzing less data.

Finding the most selective restriction typically requires statistical information extracted from previous queries or a pre-calculated histogram estimation. In our case, we explore the power and flexibility offered by a cracking DBMS. The most selective restriction can be found using the cracker indices, for they maintain knowledge about how values are spread over a map. The size of the various pieces gives the *exact* number of tuples in a given value range. Effectively, we can view a cracker index as a self-organizing histogram. In order to estimate the result size of a selection over an attribute A, any available map

in S_A can be used. In case of alternatives, the *most aligned* map is chosen by looking at the distance of its cursor to the last position of T_A. The bigger this distance, the less aligned a map is. A more aligned/cracked map can lead to a more accurate estimation. Using the cracker index of the chosen map M_{Ax}, we locate the contiguous area w that contains the result tuples. In case the predicate on A matches with the boundaries of existing pieces in M_{Ax}, the result size is equal to the size $|w|$ of w. Otherwise, we assume that w consists of n pieces $W_1, \ldots, W_n$, and derive $|w| = \sum_{i=1}^{n} |W_i|$ and $|w'| = \sum_{i=2}^{n-1} |W_i|$ as upper and lower bounds respectively. We can further tighten these bounds by estimating the qualifying tuples in W_1 and W_n, e.g., using interpolation.

One can think more directions regarding the choice of the map set, e.g., take a route that is beneficial for future queries. For example, choose a set that leads to the creation of more cracker maps or that will force alignment of existing ones that are left unaligned for a long period. However, this includes an *investment* which is not strictly in line with the cracking approach, where everything happens on demand (while processing queries), concentrating on and adapting to the current user requests, only.

Disjunctive Queries

Disjunctive queries are handled in a symmetrical way. This time the first selection creates a bit vector with size *equal* to the size of the map and not to the size of the cracked area w (as with conjunctions). The rest of the selections need to analyze the areas *outside* w for any *unmarked* tuples (and not inside w for marked tuples as with conjunctions) that might qualify and refine the bit vector accordingly. The choice of the map set is again symmetric; we choose a set based on the *least* selective attribute. In this way, the areas that need to be analyzed outside the cracked area are as small as possible. As for conjunctive queries, we benefit from having maps aligned which allows for sequential access patterns when refining the bit vector and when projecting attributes. For disjunctions, we need a set of new select operators that are used in a similar way as for conjunctions.

`sideways.dselect_create_bv(A,v1,v2,B,v3,v4)` cracks M_{AB} according to $v_1 < A < v_2$ and creates the bit vector according to $v_3 < B < v_4$ as follows.

(1)-(7) Equal to `sideways.select` in Section 5.3.2.

(**8**) Create bit vector for M_{AB} with $v_3 < B < v_4$. All tuples in area w are marked as true. The bit vector is returned as the result.

`sideways.dselect_refine_bv(A,v1,v2,B,v3,v4,bv)`
cracks M_{AB} according to $v_1 < A < v_2$ and extends an intermediate result by refining the bit vector bv as follows.

Operator `sideways.select_refine_bv(A,v1,v2,B,v3,v4,bv)`

(1)-(7) Equal to `sideways.select` in Section 5.3.2.

(**8**) By analyzing the unmarked tuples outside w, refine bit vector bv with $v_3 < B < v_4$ and return it as result.

5.3.4 Complex Queries

Until now we studied multi-selections/projections queries. The rest of the operators are not affected by the physical reorganization step of cracking as no other operator, other than tuple reconstruction, depends on tuple insertion order. Thus, joins, aggregations, groupings etc. are all performed efficiently using the original column-store operators (e.g., see our experimental analysis). In fact, more complex queries, e.g., queries with multiple non-tuple order-preserving operators like `join`, `order by`, etc., benefit significantly from sideways cracking, since tuple reconstructions after these operators are restricted to smaller areas than in a non-cracking system, leading to more cache-friendly access patterns. Potentially, many operators can exploit the clustering information in the maps, e.g., a `max` can consider only the last piece of a map or a `join` can be performed in a partitioned like way exploiting disjoint ranges in the input maps. We leave such directions for future work consideration as they go beyond the scope of this chapter.

5.3.5 Updates

Update algorithms for a cracking DBMS have been proposed and analyzed in detail in Chapter 4. An update is not applied immediately. Instead, it remains as a *pending* update and it is applied only when a query needs the relevant data assisting the self-organizing behavior. This way, updates are applied *while* processing queries and affect only those tuples relevant to the query at hand. For each cracker column, there exist a pending insertions and a pending deletions column. An update is merely translated into a deletion and an insertion. Updates are applied/merged in a cracker column without destroying the knowledge of its cracker index which offers continual reduced data access after an update.

Sideways cracking is compatible with the techniques of Chapter 4 as follows. Each map M_{AB} has a pending insertions table holding (A,B) pairs. Insertions

are handled independently and on demand for each map using the Ripple algorithm. The extension is that the first time an insertion is applied on a map of set S_A, it is also logged in tape T_A so that the rest of the S_A maps can apply the insertions in the correct order during alignment. For deletions we only need one pending deletions column for each set S_A as we only need (A,`key`) pairs to identify a deletion. Since maps do not contain the tuple `keys`, as cracker columns do, we maintain a map $M_{A\texttt{key}}$ for each set S_A. This map, when aligned and combined with the pending deletions column, gives the positions of the relevant deletes for the current query in the currently aligned maps. The Ripple algorithm is used to move deletes out of the result area of the maps used in a plan.

5.3.6 Experimental Analysis

In this section, we present a detailed experimental analysis. We compare our implementation of selection and sideways cracking on top of MonetDB, against the latest non-cracking version of MonetDB. We also compare against a presorting strategy where MonetDB can very efficiently perform selections and tuple reconstructions by exploiting oalready ordered and densely clustered data. We demonstrate that sideways cracking brings significant performance benefits and a self-organizing behavior dealing gracefully with tuple reconstruction. We use a 2.4 GHz AMD Athlon 64 processor equipped with 2 GB RAM. The operating system is Fedora Core 8 (Linux 2.6.23). Unless mentioned otherwise, all experiments use a relational table of 9 attributes (A_1 to A_9), each containing 10^7 randomly distributed integers in $[1, 10^7]$.

All experiments presented in this chapter focus on in-memory query processing. However, the problems addressed stem from the access patternsin tuple reconstruction. Our solutions in Sections 5.3 and 5.4 improve these patterns, Given that I/O favors sequential over random access patterns even more, than memory does thus we are confident that our techniques will scale well.

Exp1: Varying Tuple Reconstructions

The first experiment demonstrates the behavior in query plans with one selection, but with multiple tuple reconstructions:

$$(q_1) \quad \text{select } max(A2),\ max(A3) \ldots \text{ from R where } v_1 < A1 < v_2$$

We test with queries with 2 to 8 attributes in the select clause. For each case

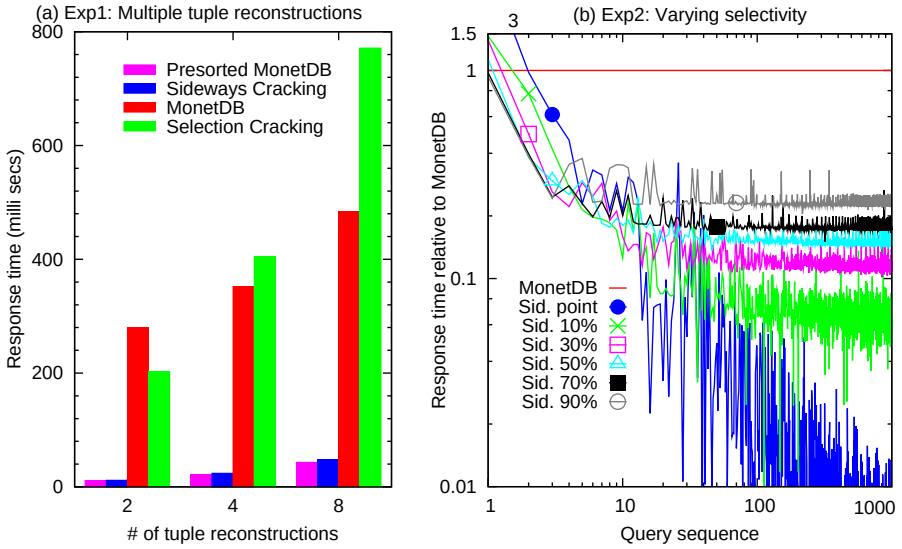


Figure 5.11: Improving tuple reconstruction

we run 100 queries requesting random ranges of 20% of the tuples. Figure 5.11(a) shows the results for the $100th$ query (full query sequence behavior is shown in next experiments). The structures in both cracking approaches have been reorganized by the previous 99 queries.

For all systems, increasing the number of tuple reconstructions increases the overall cost while presorted MonetDB and sideways cracking significantly outperform the others.

Presorted			Sid. Cracking			Sel. Cracking			MonetDB		
Tot	*TR*	*Sel*	*Tot*	*TR*	*Sel*	*Tot*	*TR*	*Sel*	*Tot*	*TR*	*Sel*
43	0.2	0.02	47	0.4	0.5	771	725	0.3	483	211	229

The above table breakes down the cost (in milli secs) for the case of 8 tuple reconstructions. It shows the contribution of tuple reconstruction (TR) and selection (Sel) to the total cost (Tot). For presorted data, the table is already sorted on the selection attribute. Naturally, selections happen very fast (using binary search). Tuple reconstructions are also extremely fast since the projection attributes are already aligned with the selection result, given that only tuple

order-preserving operators are involved in these queries (we show more complex examples later on). Sideways cracking achieves similar performance to presorted data by continuously aligning and physically clustering relevant data together both for selections and for tuple reconstructions.

On the contrary, selection cracking improves over MonetDB significantly on selections but suffers from tuple reconstruction costs. With MonetDB, the `select` operator is order-preserving, hence, tuple reconstruction is performed using in-order positional `key`-lookups into the projection attribute's base column. The resulting sequential access pattern is very cache-friendly ensuring that each page or cache-line is loaded at most once. On the contrary, with selection cracking, the result of `crackers.select` is no longer aligned with the base columns due to physical reorganization. Consequently, the tuple reconstruction is performed using *randomly ordered* positional `key`-lookups into the base column. Lacking both spatial and temporal locality, the random access pattern causes significantly more cache-/page-misses, making tuple reconstruction more expensive.

Exp2: Varying Selectivity

We repeat the previous experiment for 2 tuple reconstructions, but this time we vary selectivity factors from point queries up to 90% selectivity. We run 10^3 queries selecting randomly located ranges/points.

To make the differences more prominent, Figure 5.11(b) shows the response time relative to the performance of non-cracking MonetDB (per selectivity factor). Sideways cracking significantly outperforms MonetDB on all selectivity ranges. In general, the first query is slightly slower for sideways cracking since the appropriate maps are created. After only a few queries, the maps are physically reorganized to an extent that significantly fewer non-qualifying tuples have to be accessed, allowing sideways cracking to quickly outperform MonetDB. As queries become less selective, sideways cracking outperforms MonetDB sooner (in terms of processed queries). With less selective queries, more tuples have to be reconstructed producing higher costs for the non-cracking column-store. For the same reason, with more selective queries, the relative benefit of cracking is smaller for the initial queries as the (smaller) benefits in tuple reconstruction are being partially shadowed by the higher cracking costs of the first queries.

Let us discuss these trends in a bit more detail The costs involved are the cracking and the tuple reconstruction costs. With cracking, the queries towards the beginning of a query sequence incur a higher cost from the actual cracking actions, i.e., the first query has to create the necessary maps and has to crack the

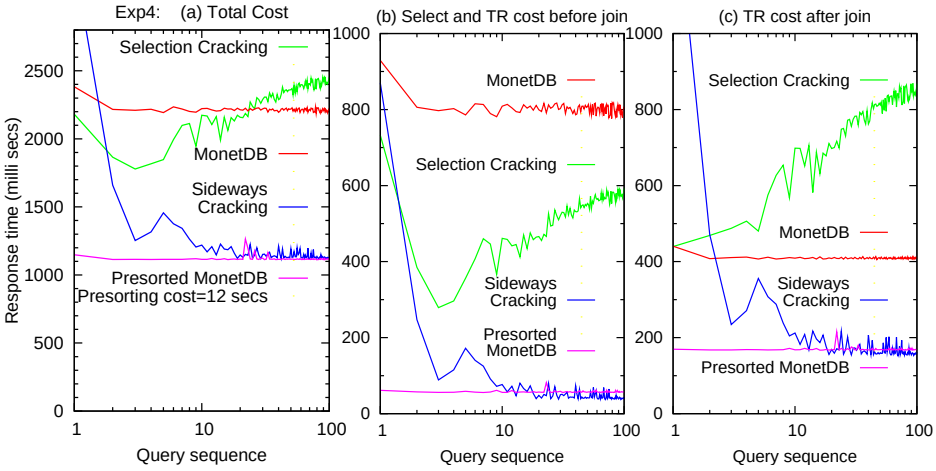


Figure 5.12: Join queries with multiple selections and tuple reconstructions (TR)

whole maps, while the next few queries have to crack (on average) big portions of these maps. As more queries arrive, the more the maps are cracked, and subsequently the less data need to be touched and cracked. This way, as the query sequence evolves, the cracking costs drop (for example, observe Figures 5.12(a), (b) and (c) where we include a break down of the costs to see this behavior).

The second cost is the tuple reconstruction cost. Naturally, with more selective queries, tuple reconstruction becomes less expensive as less tuples have to be partially reconstructed before applying an operator. This way, for more selective queries, sideways cracking has a smaller *relative* benefit to MonetDB *for the initial queries*, i.e., MonetDB spends less time in tuple reconstruction compared to less selective queries, while cracking still has to spend similar effort in cracking the maps as this is independent of the selectivity. As the query sequence evolves, the behavior reverses, i.e., once the maps are sufficiently cracked and thus the cracking costs drop, then the relative gain for sideways cracking is bigger for more selective queries.

Exp3: Join Queries

We proceed with join queries that both select and project over multiple attributes. Two tables of 7 attributes and the following query are used.

$$(q_2)\quad \begin{array}{l} \text{select } max(R1),max(R2),max(S1),max(S2) \text{ from } R,S \\ \text{where } v_1<R3<v_2 \text{ and } v_3<R4<v_4 \text{ and } v_5<R5<v_6 \\ \quad \text{and } k_1<S3<k_2 \text{ and } k_3<S4<k_4 \text{ and } k_5<S5<k_6 \\ \quad \text{and } R7=S7 \end{array}$$

We run 10^2 randomly created queries, with fixed selectivity factors of 50%, 30% and 20% for the conjunctions of each table. Figure 5.12(a) shows the results. All systems evaluate the queries starting from the most selective predicate. Sideways cracking and presorted MonetDB achieve similar performance and significantly outperform the other approaches. Presorting of course has a high preparation cost (12 secs). Figure 5.12(b) shows separately the selections and tuple reconstruction costs before the join. For a fair comparison both MonetDB and MonetDB on presorted data use the faster `rel_select` operator of selection cracking for the tuple reconstructions prior to the join. Figure 5.12(c) also shows separately the tuple reconstruction costs after the join.

For both cost components, presorted data and sideways cracking significantly improve the column-store performance by providing both very fast selections due to value ranges knowledge, but also very fast tuple reconstructions due to more efficient access patterns. Qualifying data is already aligned and clustered in smaller areas, i.e., equal to the result size of the most selective predicate. On the contrary, MonetDB and selection cracking have to reconstruct the required attributes from the full base columns.

Selection cracking loses its advantage in selections due to random access patterns in tuple reconstruction, even before the join. Also, the order of tuple`key`s in cracker columns becomes more distorted, as more queries contribute to cracking, resulting in further increasing reconstruction costs.

For all systems, tuple order in the intermediate result of the inner input is lost after the join. Thus, all systems perform tuple reconstruction for this table using random access patterns. However, plain MonetDB and selection cracking need to prompt the base columns as the tuples to be reconstructed for each attribute A are scattered through the whole base column of A. On the other hand, MonetDB on presorted data and sideways cracking have the qualifying data clustered in a smaller area in each column and thus can improve significantly by loading and analyzing less data.

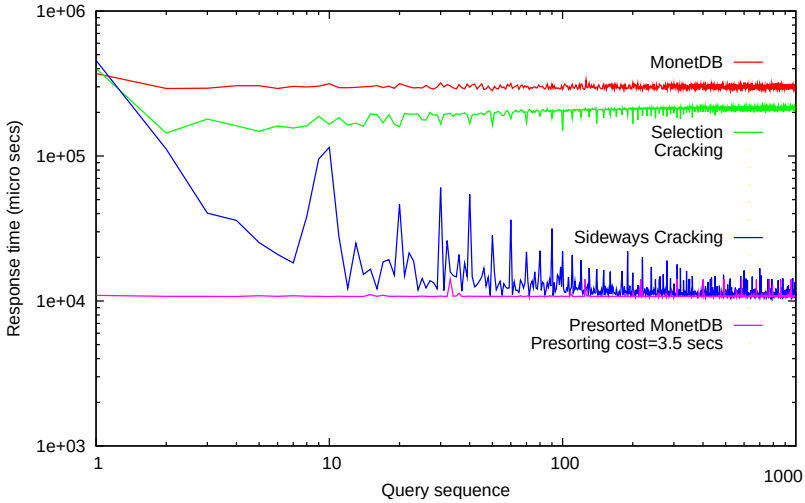


Figure 5.13: Skewed workload

Naturally, more selections or more tuple reconstructions (either before or after the join), further increase the benefits of presorted data and sideways cracking.

These results confirm that favoring sequential over random access patterns using aligned maps from only one map set and applying bit-vector filtering for the conjunctions and disjunctions is the key technique to achieve highly efficient multi-attribute queries in a cracking DBMS.

Exp4: Skewed Workload

Sideways cracking gracefully adapts to the workload and exploits any opportunity to improve performance. To illustrate this powerful property, assume a 3 attribute table and the following query type.

$$(q_3) \quad \text{select } max(B),max(C) \text{ from R where } v_1 < A < v_2$$

We choose v_1 and v_2 such that 9/10 queries request a random range from the first half of the attribute value domain, while only 1/10 queries request a random range from the rest of the domain. All queries request 20% of the

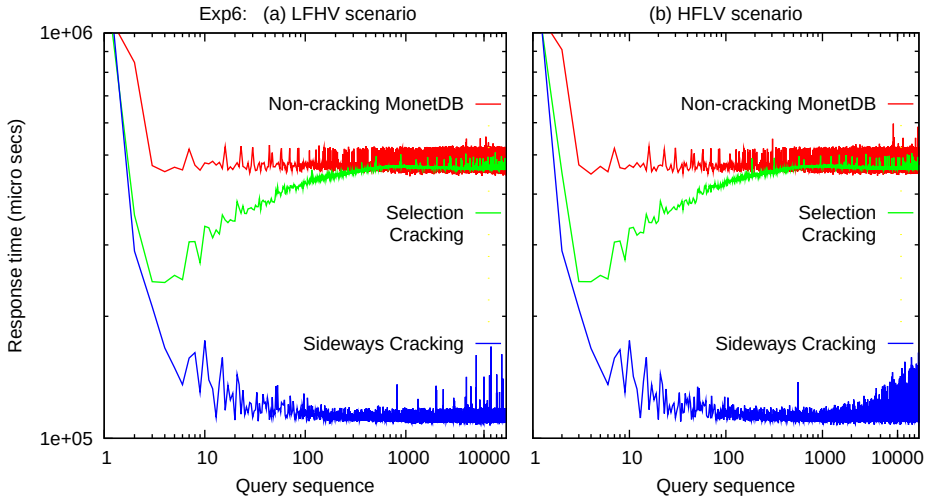


Figure 5.14: Effect of updates

tuples. Figure 5.13 shows that sideways cracking achieves high performance similar to presorted data by always using cache-friendly patterns to reconstruct tuples. Non-cracking MonetDB is not affected (experiments with smaller selectivity factors showed improvements with a skewed workload due to cache effects). Skew affects the "learning" rate of sideways cracking, making it reach the best performance quickly for the hot-set. Since most of the queries focus on a restricted area of the maps, cracking can analyze this area faster (in terms of query requests) and break it down to smaller pieces (which are faster to process). With queries outside the hot-set, we have to analyze a larger area (though not the whole column). This is why we see the peaks roughly every 10 queries. However, as more queries touch the non-hot area, in a self-organizing way, sideways cracking improves performance also for the non-hot set.

Exp5: Updates

Two scenarios are considered for updates, (a) the high frequency low volume scenario (HFLV); every 10 queries we get 10 random updates and (b) the low frequency high volume scenario (LFHV); every 10^3 queries we get 10^3 random updates. Random q_3 queries are used. Figure 6.15 shows that sideways cracking

maintains high performance and a self-organizing behavior through the whole sequence of queries and updates demonstrating similar performance as we show in Chapter 4 where the basic updates techniques were introduced. We do not run on presorted data, here, since to the best of our knowledge there is no efficient way to maintain multiple sorted copies under frequent updates in column-stores (Harizopoulos et al., 2006). This is an open research problem. Obviously, resorting all copies with every update is prohibitive.

5.4 Partial Sideways Cracking

The previous section demonstrated that sideways cracking enables a column-store to efficiently handle multi-attribute queries. It achieves similar performance to presorted data but without the heavy initial cost and the restrictions on updates and workload prediction. So far, we assumed that no storage restrictions apply. As any other indexing or caching mechanism, sideways cracking imposes a storage overhead. This section addresses this issue via partial sideways cracking. An extensive experimental analysis shows that it significantly improves performance under storage restrictions and enables efficient workload adaptation by partial alignment.

5.4.1 Partial Maps

The motivation for partial maps comes from a divide and conquer approach. The main concepts are the following.

- Maps are only *partially* materialized driven by the workload.
- A map consists of *several chunks*.
- Each chunk is a *separate* two-column table and contains a given value *range* of the head attribute of this map.
- Each chunk is treated *independently*, i.e., it is cracked separately and it has its own tape.

Figure 5.15 illustrates a simplified example of partial maps in action. It gives a quick view on the main ideas and concepts. We proceed by discussing in more detail the partial cracking architecture.

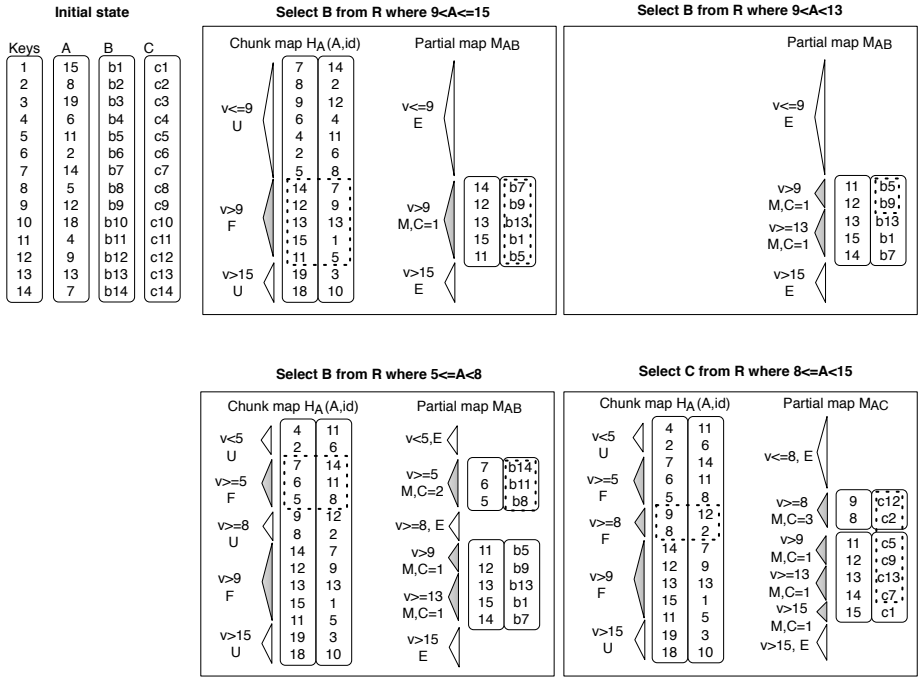


Figure 5.15: Using partial maps (U=Unfetched, F=Fetched, E=Empty, M=Materialized, C=ChunkID)

Basic Definitions

Let us now discuss in detail how sideways cracking supports partial maps. A map set S_A of an attribute A consists of (a) a collection of partial maps and (b) a *chunk map* H_A. H_A contains A values along with the respective tuple `key`. Its role is to provide the partial maps of S_A with any missing chunks when necessary. Each partial and chunk map has an AVL-tree based index to maintain partitioning information. Different maps in the same set do not necessarily hold chunks for the same value ranges. A partial map is created when a query needs it for the first time. The chunk map for a set S, is created along with the creation of the first chunk of the first partial map in S.

An area w of a chunk map is defined as *fetched* if at least one partial map has fetched all tuples of w to create a new chunk. Otherwise, w is called *unfetched*.

Similarly, an area c of a partial map is defined as *materialized* if this map has created a chunk for c. Otherwise, c is called *empty*. Figure 5.15 shows some simple examples.

For each fetched area w, the index of a chunk map maintains (i) a list of references to w, i.e., the IDs of the partial maps that currently hold a chunk created by fetching w, and (ii) a tape where all the cracks that happen on the chunks created by w are logged. If all these chunks are dropped (discussed below under "Storage Management"), then w is marked again as unfetched and its tape is removed.

Creating Chunks

New chunks for a map M_{Ax} are created on demand, i.e., each time a query q needs tuples from an empty area c of M_{Ax}. The area c corresponds to an area w of H_A. We distinguish two cases depending on whether w is fetched or not. Firstly, if w is unfetched, then currently no other map in S_A holds any chunks created from w. In this case, depending on the value range that q requires, we either make a new chunk using all tuples of w or crack w in smaller areas to materialize only the relevant area (see examples in Figure 5.15). Secondly, in case w is already marked as fetched, it must not be cracked further, as this might lead to incorrect alignment as described in Section 5.3.2. For example, if multiple maps are used by a single query q that requires chunks created from an area w, then these chunks will not be aligned if created by differently cracked instances of w. Hence, a new chunk is created using all tuples in w. To actually create a new chunk for a map M_{AB}, we use the `keys` stored in w to get the B values from B's base column.

Storage Management

A partial map is an *auxiliary* data structure, i.e., without loss of primary information, any chunk of any map can be dropped at any time, if storage space is needed, e.g., for new chunks. In the current implementation, chunks are dropped based on how often queries access them. After a chunk is dropped, it can be recreated at any time, as a whole or only in parts, if the query workload requires it. This is a completely self-organizing behavior. Assuming there is no idle time in between, no available storage, and no way to predict the future workload, this approach assures that the maximum available storage space is exploited, and that the system always keeps the chunks that are really necessary for the workload hot-set.

Before creating a new chunk, the system checks if there is sufficient storage available. If not, enough chunks are dropped to make room for the new one. Dropping a chunk c involves operations to update the corresponding cracker index I. To assist the learning behavior lazy deletion is used, i.e., all nodes of I that refer to c are not removed but merely marked as deleted and hence can be reused when c (or parts of it) is recreated in the future.

Dropping the Head Column

The storage overhead is further reduced by dropping the head column of actively used chunks, at the expense of loosing the ability of further cracking these chunks. We consider two opportunities.

First, we drop the head column of chunks that have been cracked to an extend that each piece fits into the CPU cache. In case a future query requires further cracking of such pieces, it is cheap to sort a piece within the CPU cache. This action is then logged in the tape to ensure future alignment with the corresponding chunks of other maps.

Second, we drop the head column of chunks that have not been cracked recently as queries use their pieces "as is". Once we need to crack such a chunk c in the future, we only need to recover and align the head as follows. If a chunk c' (of the same area as c) in an other map still holds the head and is less or equally aligned to the state that c was when it lost its head, then the head is recovered from c'. Otherwise, the head is taken from the chunk map. The first case is cheaper as less effort is needed to align the head.

Chunk-wise Processing

As seen in Section 5.3.3, each sideways cracking operator O first collects (using proper cracking and alignment) in a contiguous area w of the used map M all qualifying tuples based on the head attribute of M. Then, it runs the specific operator O over the area w, e.g., create or refine a bit vector based on a conjunctive predicate, perform a projection, etc. With partial sideways cracking we have the opportunity to improve access patterns even more by allowing chunk-wise processing. Each operator O handles one chunk c of a map at a time, i.e., load c, create c if the corresponding area is empty, crack or align c if necessary and finally run O over c.

An operator goes through the relevant chunks of a map. On its way, it may need to align and or crack one or more chunks while any missing ones will be created. The core algorithm used for all operators is described in Algorithm 5

Algorithm 5 BasicPartial(Map,Lval,Hval,Operator)
Run "Operator" for tuples of "Map" with head values in [Lval,Hval]

```
 1: foundL,LNode=SearchCrackerIndex(Map,Lval)
 2: foundH,HNode=SearchCrackerIndex(Map,Hval)
 3: found=foundL and foundH
 4: cmfoundL,ChunkMapNode=SearchCrackerIndex(ChunkMap(Map),Lval)
 5: cmfoundH=SearchCrackerIndex(ChunkMap(Map),Hval)
 6: chunk=LNode.chunk
    {Both bounds fall within the same materialized chunk}
 7: if chunk.isMaterialized==true and chunk=HNode.chunk then
 8:    found,Lpos,Hpos=Sync(chunk,Lval,Hval,found,Full)
 9:    if found==true then Lpos,Hpos=Crack(chunk,Lval,Hval)
10:    run(Operator,chunk,Lpos,Hpos)
11:    return

    {The bounds fall in different chunks. Start from the chunk in the low bound}
12: if chunk.isMaterialized==true then
13:    foundL,Lpos=Sync(chunk,Lval,null,foundL,Full)
14:    if foundL==true then Lpos=Crack(chunk,Lval,null)
15: else
16:    area=ChunkMapNode.area
17:    if CBNode.IsFetched==true then
18:       chunk=NewChunk(area,0,area.LastPos)
19:       foundL,Lpos=Sync(chunk,Lval,null,foundL,Full)
20:       if foundL==true then Lpos=Crack(chunk,Lval,null)
21:    else
22:       if cmfoundL==true then Lpos=Crack(area,Lval,null)
23:       chunk=NewChunk(area,Lpos,area.LastPos)
24:       Lpos=0
25: run(Operator,chunk,Lpos,chunk.LastPos)

26: BasicPartialMainPieces(Operator,ChunkMapNode,LNode,Hval)
```

and 6 supported by Algorithm 7 for the alignment part. Let us go through its main points. First, we search the index of the relevant partial map for qualifying chunks. Function *SearchCrackerIndex* fills variables *LNode* and *HNode* with the AVL-tree nodes of the index where the *Lval* and *Hval*, respectively, fall. If both bounds fall within the same materialized chunk c, we need to align and crack c (alignment will be discussed in the next paragraph in more detail). The Crack function will run only if variable *found* is *false* which means that the requested

Algorithm 6 BasicPartialMainPieces(Operator,ChunkMapNode,LNode,Hval)
Run "Operator" for all remaining pieces until outside Hval.

```
 1: done=false
 2: while done==false do
 3:    ChunkMapNode=ChunkMapNode.nextArea
 4:    area=ChunkMapNode.area
 5:    LNode=LNode.nextChunk
 6:    chunk=LNode.chunk

       {Each chunk is only partially aligned.}
       {Only the last chunk is fully aligned if we cannot avoid cracking.}
 7:    if chunk.isMaterialized==true then
 8:       if Hval<chunk.hgh then
 9:          foundH,Hpos=Sync(chunk,null,Hval,foundH,Full)
10:          if foundH==true then Hpos=Crack(chunk,null,Hval)
11:          Lpos=0
12:          done=true
13:       else
14:          Sync(chunk,null,null,foundH,Partial)
15:          Lpos=0, Hpos=chunk.LastPos
16:    else
17:       if area.IsFetched==true then
18:          chunk=NewChunk(area,0,area.LastPos)
19:          foundH,Hpos=Sync(chunk,null,Hval,foundH,Partial)
20:          if Hval<chunk.hgh then
21:             if foundH==false then
22:                foundH,Hpos=Sync(chunk,null,Hval,foundH,Full)
23:             if foundH==true then Hpos=Crack(chunk,null,Hval)
24:             Lpos=0
25:             done=true
26:       else
27:          if Hval<area.hgh then
28:             if cmfoundH==true then Hpos=Crack(area,null,Hval)
29:             chunk=NewChunk(area,0,Hpos)
30:             done=true
31:          else
32:             chunk=NewChunk(area,0,area.LastPos)
33:          Lpos=0, Hpos=chunk.LastPos
34:    run(Operator,chunk,Lpos,Hpos)
```

bounds have not been used in the past. It cracks c such that all qualifying tuples are in a contiguous area w. Then, if necessary, it updates the index of *Map*, the tape and the cursor of c. It also fills variables *Lpos* and *Hpos* with the boundary positions of w. Then, the requested *Operator* runs over the qualifying tuples of c. If the bounds fall in different chunks, we start from the chunk where the lower bound *Lval* falls. The algorithm handles all relevant chunks one by one, until it hits a chunk with values higher than the upper bound *Hval*. If we hit an empty area, then a new chunk is created as discussed above.

It handles all relevant chunks one by one, until it hits a chunk with values higher than the upper bound *Hval*. If we hit an empty area, then a new chunk is created as discussed above.

Partial Alignment

Partial maps allow significant optimizations during alignment. The key observation is that we do not always need to perform *full* alignment, i.e., align a chunk c up to the last entry of its tape. If c is not going to be cracked, it only needs to be aligned with respect to the corresponding chunks of the other maps of the same map set used in this query, i.e., up to the *maximum* cursor of these chunks. We call this *partial* alignment. As seen in Algorithm 5, when performing any operator over a map, only the *boundary* chunks might need to be cracked, i.e., the first chunk where the lower bound falls and the last chunk where the upper bound falls; *all* other chunks in between benefit from partial alignment.

Even for boundary chunks, partial sideways cracking can in many cases avoid full alignment as follows. Assume a chunk c as a candidate for cracking based on a bound b. First, we perform partial alignment on c and *monitor* the alignment bounds. If b matches one of the past cracks, then cracking and thus full alignment of c is not necessary. Otherwise, full alignment starts. However, even then, if b is found on the way, then alignment stops and c is not cracked (see Algorithms 5 and 7 for a step by step description). Since every operator runs separatelly, i.e., it cannot have the knowledge of which maps the current query requires, we introduce the follwoing simple extension in the query plans for partial maps. Before the normal query plan starts a series of operators $registerMap(A1, A2)$ indicate that the partial map M(A1,A2) will be used in this query. In this way all necessary maps are registered. They are simply added in a list in the cracker column of the respective map set. After the normal query plan is done, a single operator $clearReferences(A1)$ removes all references of map set $A1$.

Algorithm 7 Sync(chunk,Lval,Hval,found,Mode)
Synchronize this "chunk" in Partial or Full "Mode". Monitor the alignment bounds in case of a match with "Lval" and "Hval".

```
 1: cursorPos=chunk.cursor
 2: tape=chunk.tape
 3: stop=getMaximumCursorOfMapsInCurrentQuery()
 4: curMode=Mode

 5: if cursorPos==tape.size or (Mode==Partial and cursorPos==stop) then
 6:    return found

 7: while true do
 8:    low=tape[cursor].low
 9:    hgh=tape[cursor].hgh
10:    pos1,pos2=Crack(chunk,low,hgh,false)
11:    if low==Lval and hgh==Hval then
12:       Lpos=pos1
13:       Hpos=pos2
14:       found=true
15:    cursor+=1
16:    if found==true and curMode==Full then break
17:    if cursor==stop then
18:       if Mode==Full and curMode==Partial and found==false then
19:          curMode=Full
20:          stop=tape.size
21:       else
22:          break

23: chunk.cursor=stop
24: return found,Lpos,Hpos
```

Updates

The Ripple algorithm of (Idreos et al., 2007b) is already designed to update only the parts (value ranges) necessary for the running query. Thus, the update strategy and performance in partial maps remains the same as in Section 5.3. Chunk maps are treated in the same way, i.e., a chunk map H_x has its own pending updates structures and areas on H_x are updated only on demand. Thus, before making a new chunk from an unfetched area w, w is updated if necessary. Naturally, updates applied in a chunk map are also removed from

the pending updates of all partial maps in this set.

5.4.2 Experimental Analysis

We proceed with a detailed assessment of our partial sideways cracking implementation on top of MonetDB. Using the same platform as in Section 5.3.6, we show that partial maps bring a significant improvement and a self-organizing behavior under storage restrictions and during alignment.

For storage management with full maps, we use the same approach as for partial maps, i.e., existing maps are only dropped if there is not sufficient storage for newly requested maps. We always drop the least frequently accessed map(s).

For the experiments throughout this section we use a relation with 11 attributes containing 10^6 tuples and 5 multi-attribute queries (Q_i, $i \in \{1, \ldots, 5\}$) of the following form.

$$(Q_i) \quad \text{select } C_i \text{ from R where } v_1 < A < v_2 \text{ and } v_3 < B_i < v_4$$

All queries use the same A attribute but different B_i and C_i attributes, i.e., each query requires two different maps. A fully materialized map needs 10^6 tuples. All queries select random ranges of S tuples. We run 10^3 queries in batches of 100 per type, i.e., first 100 Q_1 queries, then 100 Q_2 queries, and so on, while enforcing a storage threshold of T tuples.

Handling Storage Restrictions

We use $S = 10^4$ and three different storage restrictions:

- (a) no limit. In practice, all 10 maps used by the 5 queries fit within $T = 10^7$.
- (b) $T = 6.5 * 10^6$, i.e., slightly more than required to keep 6 full maps concurrently.
- (c) $T = 2 * 10^6$, i.e., only 2 full maps can co-exist; the minimum to run one query using full maps.

Figures 5.16(a), (b) and (c) show the per query cost for each case separately. In all three plots, full maps show the same pattern. Once every 100 queries, very high peaks (i.e., per query costs) severely disturb the otherwise good performance. These peaks relate to the workload changes between query batches.

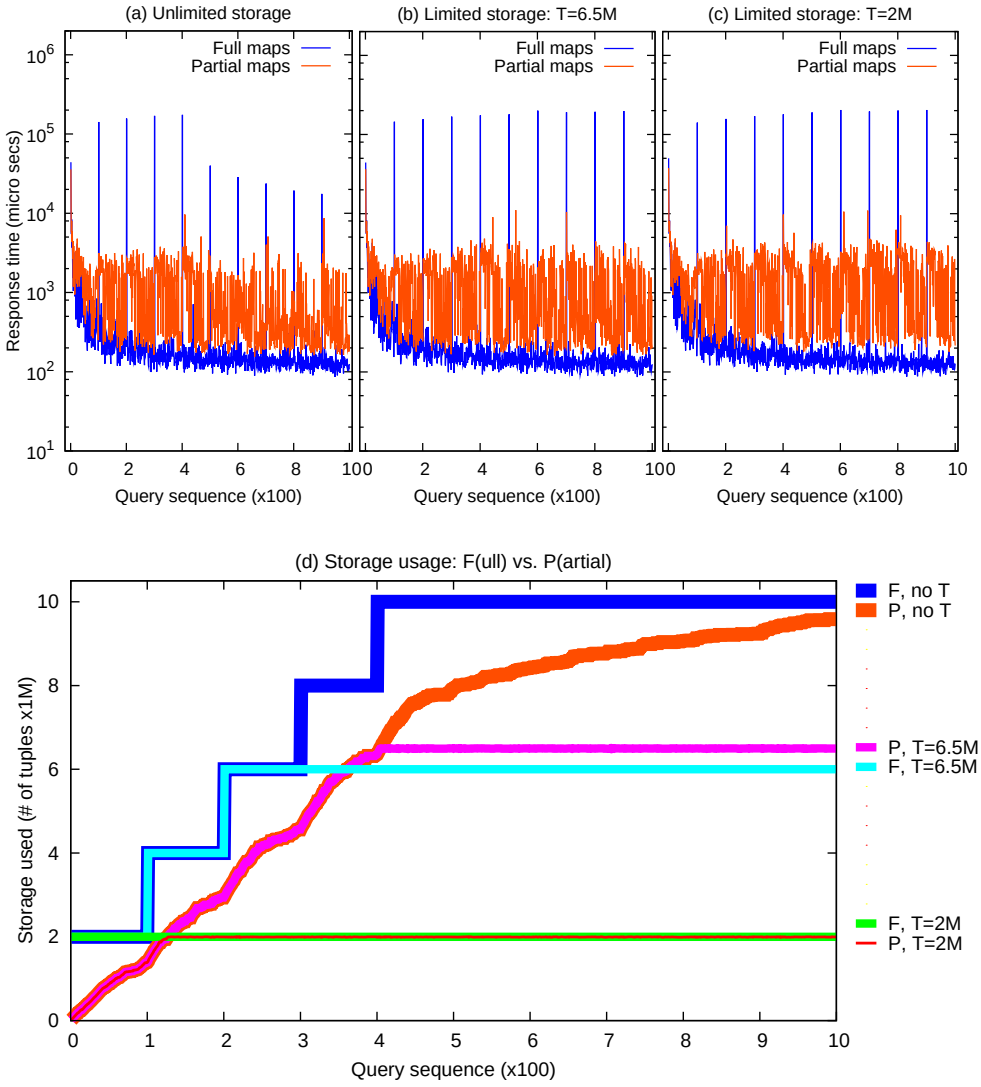


Figure 5.16: Efficient handling of storage restrictions with partial maps (S=10K)

The first 5 peaks reflect the costs of initially creating the cracker maps for each batch, plus aligning them with the cracks of the preceding batch. Requiring no alignment, the first peak is smaller than the next 4. As of query 500, the batch cycle is repeated. With unlimited storage, all created maps are still available for reuse, requiring only alignment but no recreation with peaks 5–10 in Figure 5.16(a). With limited storage, the first maps had to be dropped to make room for later batches, requiring complete recreation of the maps once the cycle restarts. Figure 5.16(d) shows full maps allocating storage in blocks of two full maps per batch.

In contrast, partial maps do not penalize single queries, but distribute the map creation and alignment costs evenly across all queries, using chuck-wise granularity to more accurately adapt to changing workloads. Due to slightly increased costs for managing individual chunks instead of monolithic maps, partial maps do not quite reach the minimal per query cost of full maps. However, this investment results in a much smoother and more predictable performance behavior due to more flexible storage management (cf., Fig. 5.16(d)). Additionally, partial maps reduce the overall costs of query sequences (cf., Fig. 5.18 & 5.19 and discussion below).

Adaptation

Partial maps can fully exploit the workload characteristics to improve performance. To demonstrate this, we re-run the basic experiment with two variations: (a) we keep the uniform workload, but increase the selectivity using $S = 10^3$; (b) we keep $S = 10^4$, but use a skewed workload. To simulate the skew, we force 9/10 queries to request random ranges from only 20% of the tuples while the remaining queries request ranges from the rest of the domain. Both runs use a storage threshold of $T = 6.5{*}10^6$.

Figures 5.17(a) and (b) depict the results. Compared to the previous experiment, the workload is now focused on specific data parts, either by more selective queries or by skew. In both cases, partial sideways cracking shows a self-organizing and normalized behavior without penalizing single queries as full maps do. Being restricted to handling complete maps (holding mostly unused data), full maps cannot take advantage of the workload characteristics and suffer from lack of storage. Figure 5.17(c) illustrates that full maps demand more storage and thus quickly hit the threshold. In contrast, partial maps exploit the available storage more efficiently and more effectively by materializing only the required chunks.

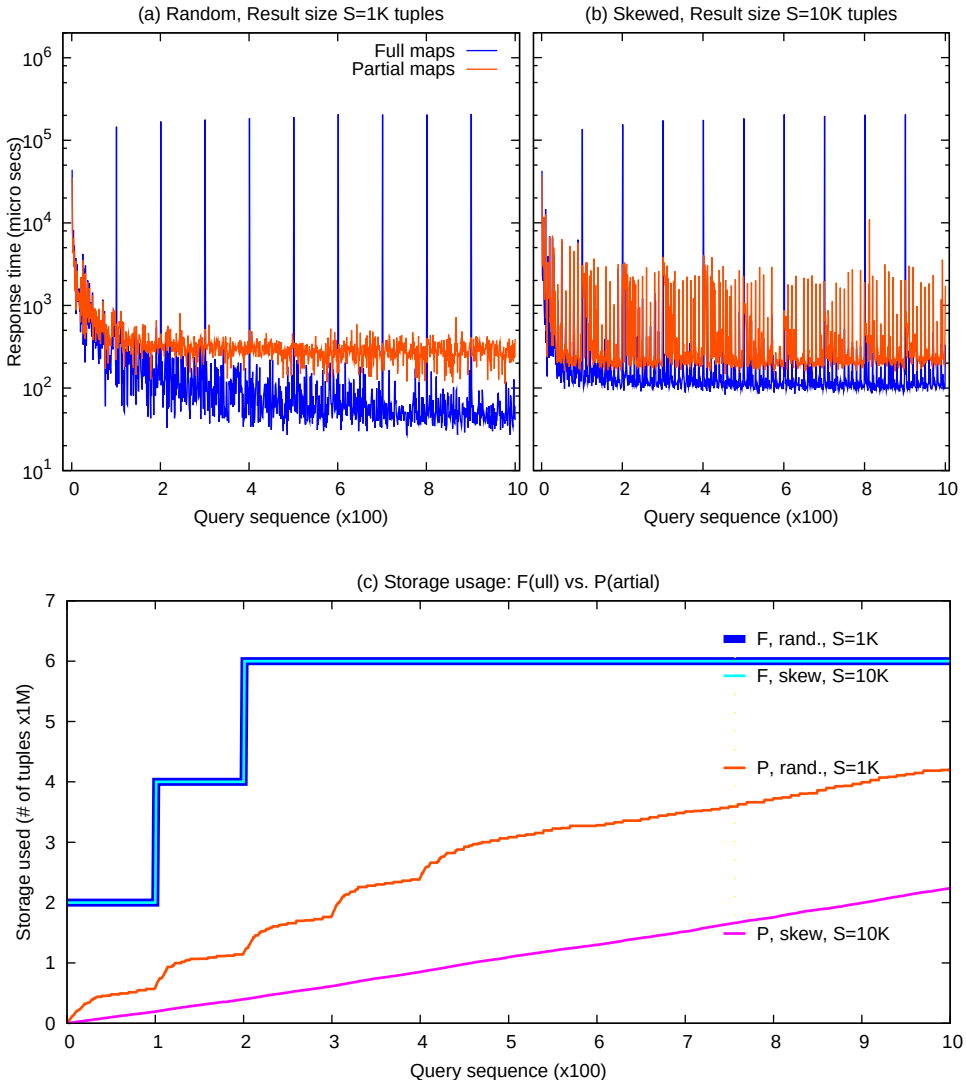


Figure 5.17: Efficient adaptation to the workload with partial maps (T=6.5M)

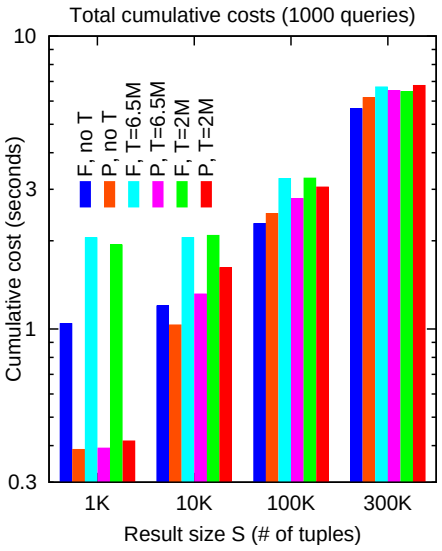


Figure 5.18: No overhead

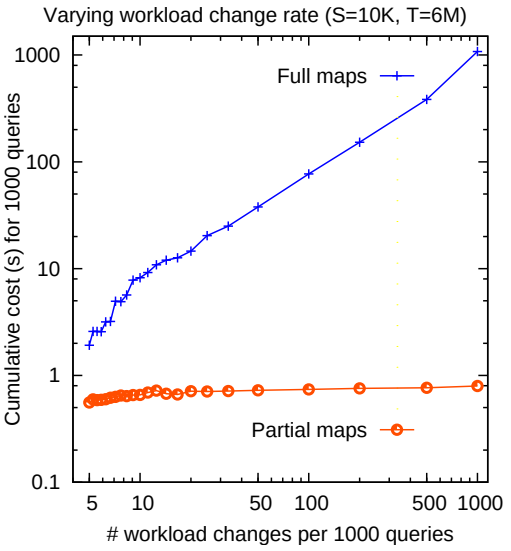


Figure 5.19: Total costs

No Overhead in Query Sequence Cost

So far, we demonstrated that partial maps provide a more normalized per query performance compared to full maps. In addition, Figure 5.18 shows that these benefits come for free. It depicts the total cost to process *all* queries in the basic experiment by varying both the selectivity and the storage threshold. With 30% selectivity ($S = 3*10^5$), both approaches have similar total cost while with more selective queries partial maps significantly outperform full maps. This behavior combined with the more normalized per query performance gives a strong advantage to partial maps. The next experiment demonstrates that the advantage of partial maps over full maps increases with more frequent workload changes.

Adapting to Frequently Changing Workloads

In all previous experiments we assume a fixed rate of changing workload, i.e., every 100 queries. Here we study the effect of varying this parameter. We run the basic experiment with fixed $S = 10^4$ and $T = 6*10^6$ but for various

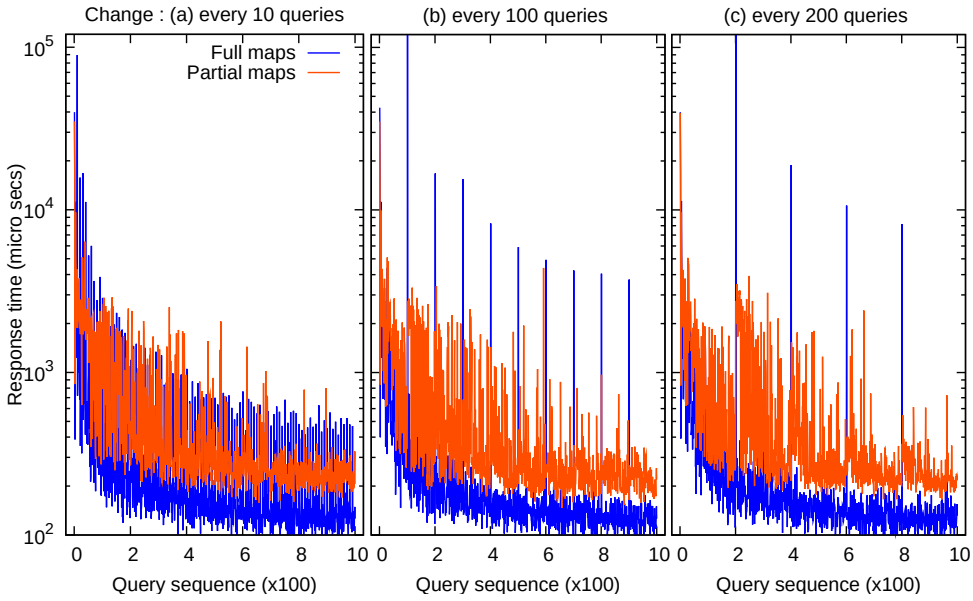


Figure 5.20: Improving alignment with partial maps (S=10K, T=unlimited)

different rates of changing the workload. Figure 5.19 shows the total cost to process all queries for each case. The performance of full maps faces a significant degradation as the workload changes more often, causing maps to be dropped and recreated more frequently. In contrast, due to flexible and adaptive chunk management, partial maps offer a stable high performance that decreases hardly with more frequent workload changes.

Alignment Improvements

Let us now demonstrate the benefits of partial maps during alignment. We run the basic experiment for $S = 10^4$. To concentrate purely on the alignment cost we use only two types of queries and assume no storage restrictions. Figure 5.20 shows results for changing the workload every 10, 100 or 200 queries. As we decrease the rate of changing workloads, the peaks for full maps become less frequent, but higher. These peaks represent the alignment cost. Each time the workload changes, the maps used by the new batch of queries have to be

aligned with the cracks of the previous batch; the longer the batch, the more cracks, the higher the alignment costs. Partial maps do not suffer from the alignment cost. Being able to align chunks only partially, and only those required for the current query, partial maps avoid penalizing single queries, bringing a smoother behavior to the whole query sequence. Furthermore, notice that as more queries are processed, partial maps gain more information to continuously increase alignment performance, assisting the self-organizing behavior.

5.5 TPC-H experiments

In this section, we evaluate our implementation in real-life scenarios using the TPC-H benchmark (TPC-H, 2009) (scale factor 1) on the same platform as in the previous experiments. We use the TPC-H queries that have at least one selection on a non-string attribute, i.e., Queries 1, 3, 4, 6, 7, 8, 10, 12, 14, 15, 19, & 20 (cf., (TPC-H, 2009)). String cracking and base table joins exploiting the already partitioned cracker maps are expected to yield significant improvements also for the remaining queries, but these are directions of future work and complementary to this chapter. For each query, we created a sequence of 30 parameter variations using the random query generator of the TPC-H release. For experiments on presorted data, we created copies of all relevant tables such that for each query there is a copy primarily sorted on its selection column and (where applicable) sub-sorted on its group-by and/or order-by column. We use MySQL to show the effects of using presorted data on a row-store.

Figure 5.23 shows the costs for each query sequence. Sideways cracking achieves similar performance to presorted MonetDB (ignoring the presorting cost). Depending on the query, the presorting cost is 3 to 14 minutes, while as seen in Figure 5.23 the first sideways cracking query (in each query sequence) is between 0.75 to 3 seconds. In a self-organizing way, sideways cracking continuously improves performance without requiring the heavy initial step of presorting and workload knowledge. For most queries, it outperforms plain MonetDB as of the second run; for Queries 1 & 10, already the first run is faster.

Table 5.1 summarizes the benefits of sideways cracking ($SiCr$) and presorted MonetDB ($PrMo$) over plain MonetDB on the tested TPC-H queries (Q). Having both efficient selections and tuple reconstructions, both sideways cracking and presorted MonetDB manage to significantly improve over plain MonetDB especially for queries with multiple tuple reconstructions on large tables, e.g., Queries 1, 6, 7, 15, 19, 20. Queries with multiple non tuple-order-preserving operators (group by, order by, joins) and subsequent tuple reconstructions yield

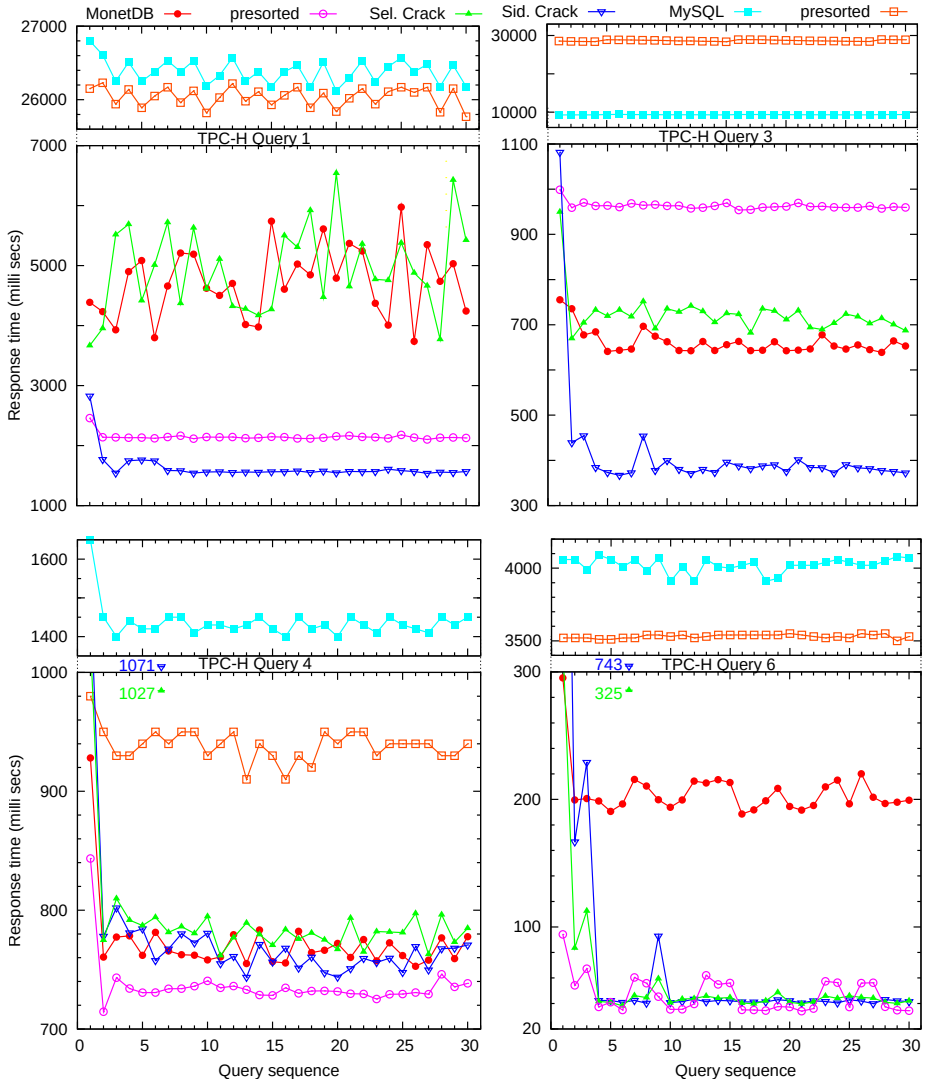


Figure 5.21: TPC-H results for Queries 1, 3, 4, 6 ("presorted" times exclude presorting costs; Q4: 3 min.; Q1,6: 11 min; Q3: 14 min.)

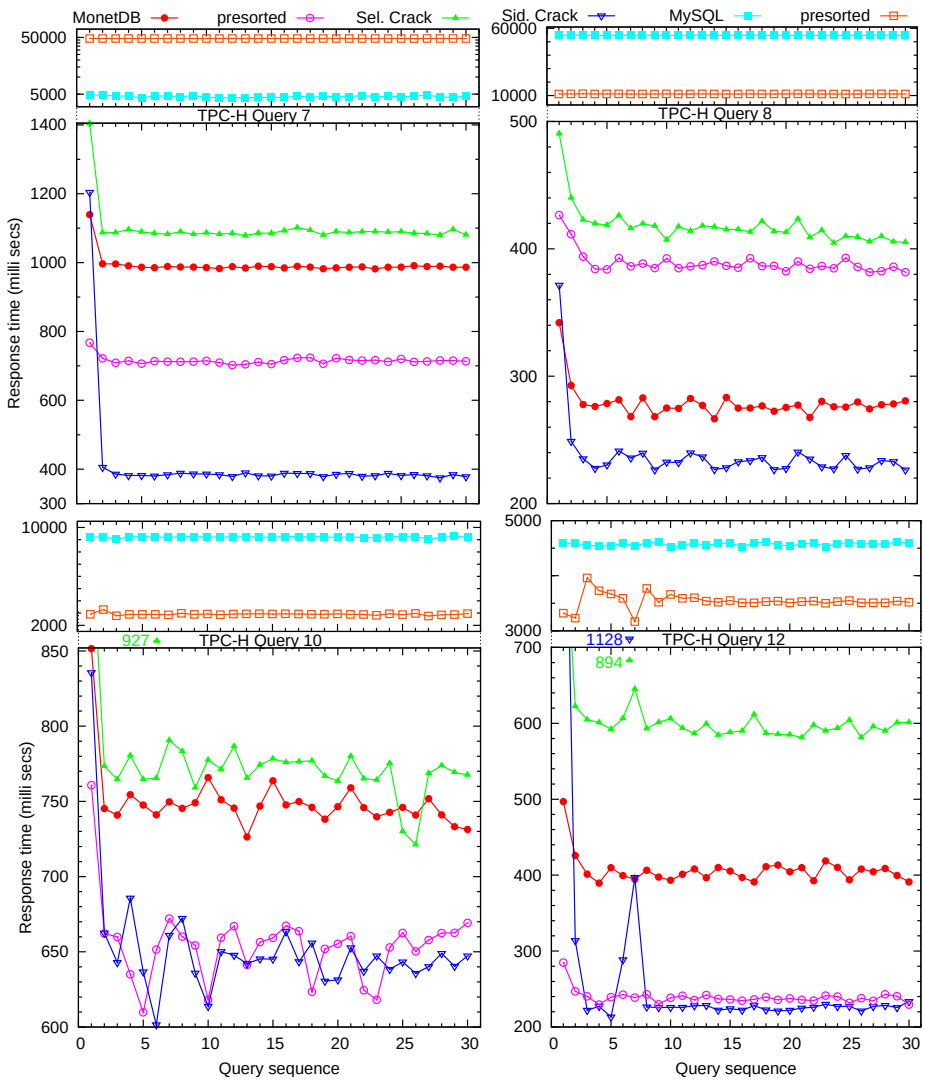


Figure 5.22: TPC-H results for Queries 7, 8, 10, 12 ("presorted" times exclude presorting costs; Q8,10: 3 min.; Q7,12: 11 min.)

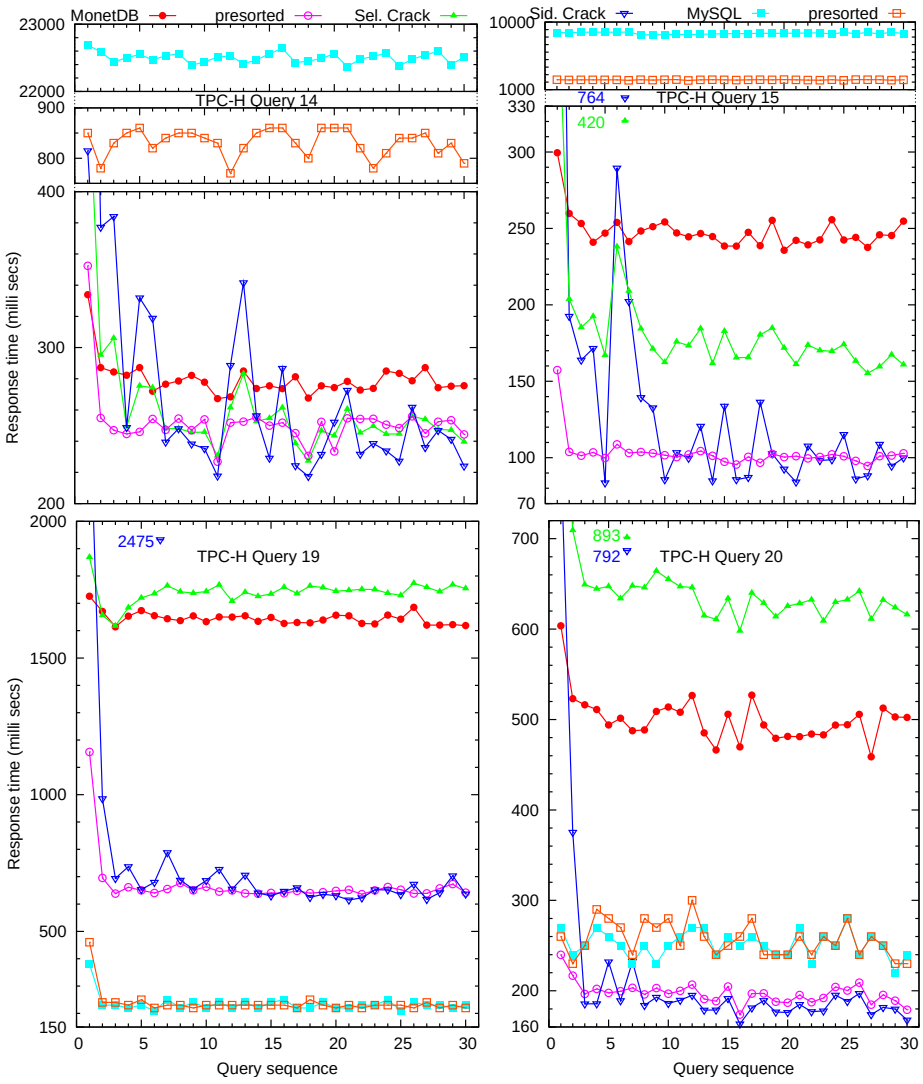


Figure 5.23: TPC-H results for Queries 14, 15, 19, 20 ("presorted" times exclude presorting costs; Q14,15,19,20: 11 min.)

	All systems						Savings	
Q	*SiCr*	*PrMo*	*Mo*	*SelCr*	*MS*	*PrMS*	*SiCr*	*PrMo*
1	1568	2127	4241	5424	26180	25770	64%	50%
3	372	959	653	687	9380	28890	44%	-46%
4	751	738	777	784	1450	940	4%	6%
6	41	34	199	42	4070	3530	80%	83%
7	377	713	986	1080	4610	47760	62%	28%
8	226	381	280	405	54870	11200	20%	-36%
10	647	669	731	767	9210	2950	12%	9%
12	233	229	390	601	4590	3520	41%	42%
14	224	244	275	239	22510	790	19%	12%
15	99	102	254	160	8470	2300	62%	60%
19	636	641	1618	1757	230	220	61%	61%
20	167	179	502	616	240	230	67%	65%

Table 5.1: TPC-H

significant gains by restricting tuple reconstructions to small column areas, e.g., Queries 1, 3, 7. Query 19 is an example where a significant amount of tuple reconstructions are needed as it contains a complex disjunctive where clause. The column-store has to reconstruct each attribute multiple times to apply the different predicates whereas the row-store processes the tables tuple-by-tuple. Sideways cracking minimizes significantly this overhead providing a comparable performance to the row-store.

In some cases using presorted data becomes slower even when compared to simple MonetDB, e.g., for Queries Q3 and Q8. In other cases cracking is considerably faster, e.g., for Query Q7. First, observe that similar behavior we see for MySQL, e.g., for Queries Q3 and Q7. The TPC-H data comes already presorted on the keys of the `Order` table. Plain MonetDB (and MySQL) exploit the sorted keys especially during joins (most queries join on `Order` keys). Extensive analysis, tracing and comparing all different query plans and the actual time needed by each operator, tracked down this behavior i.e., the extra cost with presorted data, to the hash join operator used to perform the join for the `Order` table in these queries. Given that the TPC-H data comes presorted on `Order` keys, any sorting or cracking based on different attributes destroys this initial order. For presorted data, we presorted the TPC-H data "optimally" for each query taking into account the selections it needs, and then any group by, order by actions etc. The extra cost for the presorted data runs comes from the hash join used

and more specifically the significant cost part was in prompting the hash table. This becomes far more expensive for the presorted data than in the sideways cracking case which *partially maintains* the initial order in these columns. This is purely an access pattern issue and represents an optimization opportunity; optimize for the join or for the selection and the tuple reconstructions.

At a first glance, one would expect that the same behavior should be seen for cracking as more queries crack the columns, i.e., the more we crack a column, the closer we get to the state of a fully sorted column. However, given the workload, the columns are not extensively cracked. The variation on the alternative predicates used in the TPC-H queries (as created by the TPC-H query generator) is quite limited. The effect is that cracking reaches its best performance quite fast and it does not go into access pattern issues during the joins. The optimization issue of course for cracking here is that in the long run we would like to identify these cases and stop cracking after some point to avoid such costs.

Another interesting observation is that in some TPC-H queries cracking is faster than MonetDB even for the first query (Q1 and Q10), while in some other queries it has almost the same performance (Q7 and Q8). This is surprising at first since we expect that cracking is always a bit slower for the first query as it invests in creating the necessary (missing) maps and cracking full columns. However, there are benefits that cracking always gets independent of the query, i.e., it does not have to perform expensive joins for tuple reconstruction and it does not have to materialize intermediate results after a selection or after a tuple reconstruction. The latter has an even bigger effect when only one selection per relation is involved in the query, since otherwise a bit-vector is used to filter the intermediates (see Section 5.3.3). Then, this bit vector is materialized (once) and is on-the-fly "refined" during selections and tuple reconstructions. The effect we see in the referenced queries, is mainly due to the large number of tuple reconstructions involved in these query plans.

Let's see Q1, that has the most prominent differences, in more detail. This is a single relation query with a single selection and multiple tuple reconstructions. When tracing the query plan for the fastest Q1 query of plain MonetDB (3.6 seconds), we see that MonetDB spends roughly 1.9 seconds in `join` operators for tuple reconstruction (in total there are 6 such calls in each query plan) plus 0.1-0.2 seconds for the selection. The tuple reconstructions in Q1 are too slow since they have to use random access patterns (on the whole columns) as they follow a group by and an order by operator. Depending on the predicates of the selection, the size of the intermediate result may be bigger leading to even bigger costs. On the contrary, for cracking the "investment" during the first query is

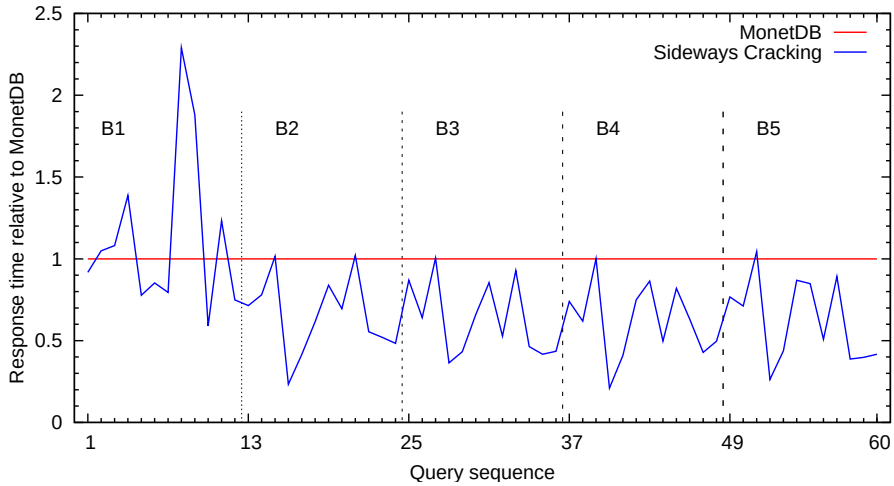


Figure 5.24: Mixed TPC-H queries

∼1.3 seconds. This includes creating and cracking all necessary maps while at the same time computing all selections and reconstructions. This investment is independent of the selectivity. In addition, by collecting the result data in small contiguous areas in the maps, reconstructions are not severely affected by random access patterns, allowing cracking to have an immediate benefit. The rest of the plans have similar costs for both cracking and MonetDB (∼1.5 seconds).

Our final experiment features a mixed workload. We run 5 sequential batches (B1..B5) of 12 different TPC-H queries with varying parameters. Figure 5.24 shows the performance of sideways cracking relative to MonetDB. Already within the first batch (B1), sideways cracking outperforms MonetDB in many queries. This is because queries can reuse maps and partitioning information created by different queries over the same attributes. The high peak in the first batch comes from Query 12 that uses a map set not used by any other query. Naturally, after the first batch sideways cracking improves even more.

5.6 Summary

In this chapter, we introduce partial sideways cracking, a key component in a self-organizing column-store based on physical reorganization in the critical path of query execution. We show that multi-attribute queries impose a significant tuple reconstruction cost due to the expensive random access patterns involved. Partial sideways cracking allows efficient tuple reconstruction using only cache-friendly sequential access patterns. It yields significant performance gains even under storage restrictions and updates. It enables efficient processing of complex multi-attribute queries by minimizing the costs of late tuple reconstruction, achieving performance competitive with using presorted data, but requiring neither an expensive preparation step nor a priori workload knowledge. With its flexible and adaptive chunk-wise architecture it yields significant gains and a clear self-organizing behavior even under random workloads, storage restrictions, and updates.

Database cracking has only scratched the surface of this promising direction for self-organizing DBMSs. The next chapter gives a nice example of how cracking can be exploited in more areas of a database kernel by introducing a new class of join algorithms that can exploit and also enhance the cracking knowledge.

Chapter 6

Crack Joins*

6.1 Introduction

The previous chapters have discussed basic cracking issues. Especially with the design of the sideways cracking architecture, we have a solid platform in our hands to study cracking in a larger variety of scenarios. Now the benefits can be seen in heavy and complex SQL queries, successfully transferring the self-organizing behavior all the way through a query plan.

In this chapter, we make a first step in demonstrating how we can expand the cracking benefits in even more operators. The continuous physical reorganization brings knowledge regarding how data is stored. This chapter studies the problem of how to *exploit* and even *enhance* this knowledge during *join* processing.

6.1.1 Contributions

We propose a new class of join algorithms, called *crack joins*, that are fully compatible with previous work on cracking and push its self-organizing behavior one significant step further. The crack joins build on top of the experience gained from both cracking and the extensive literature on join algorithms.

We present a series of crack join algorithms in a step-wise fashion. Our first algorithm purely exploits the partitioning knowledge gained from crack

*The material in this chapter has been the basis for a paper submitted for publication entitled "Adaptive Joins for Adaptive Kernels" (Idreos et al., 2010b).

operators in previous queries and avoids joining pieces with non-overlapping value ranges. Then, we discuss more advanced algorithms that, in addition to exploiting existing knowledge, they also incorporate on-the-fly *dynamic physical reorganization* of the join inputs in a way that speeds-up not only the current join but also any kind of subsequent crack operator (i.e., joins, selects or tuple reconstructions) on the same inputs even across different queries.

While processing the various pieces during a join, the inputs are physically reorganized to *align* the pieces (in terms of values ranges) across the two inputs. This makes it easier for the current operator to join the pieces but also adds knowledge to the system on how data is organized which can then be exploited by future operators.

In addition to alignment, a more *active* algorithm, physically reorganizes the inputs for reducing piece sizes also, compensating for a possible lack of "enough" past crack selections on these columns.

We also demonstrate that unbound continuous physical reorganization decreases join performance during long query sequences since the more pieces a crack join has to handle, the more administrative costs are involved. We present a cache conscious variant of the crack join that processes *groups* of pieces at a time and avoids cracking (for alignment) for every single piece. It automatically makes sure that a group of pieces to be processed optimally fits in the cache so that the pure join cost is also kept low effectively balancing administrative and cracking costs with join costs. It successfully maintains optimal performance through long query sequences and continuously changing workloads.

Finally, we show how to efficiently maintain and exploit, during cracking and updates, possible structures, e.g., hash tables, built on top of the column pieces within crack joins.

We present in detail the new algorithms and the research space. Crack joins are implemented in MonetDB. A detailed experimental evaluation shows that they bring a significant performance benefit and a clear self-organizing behavior.

6.1.2 Outline

The rest of this chapter is organized as follows. Section 6.2 presents the basic crack join techniques and algorithms along with a detailed experimental analysis. Section 6.3 presents the need for a cache conscious variant of the crack join and the new algorithm with a thorough experimental evaluation. Section 6.4 shows how to exploit crack joins even when no or just a few selections contribute to cracking. and finally Section 6.5 concludes the chapter.

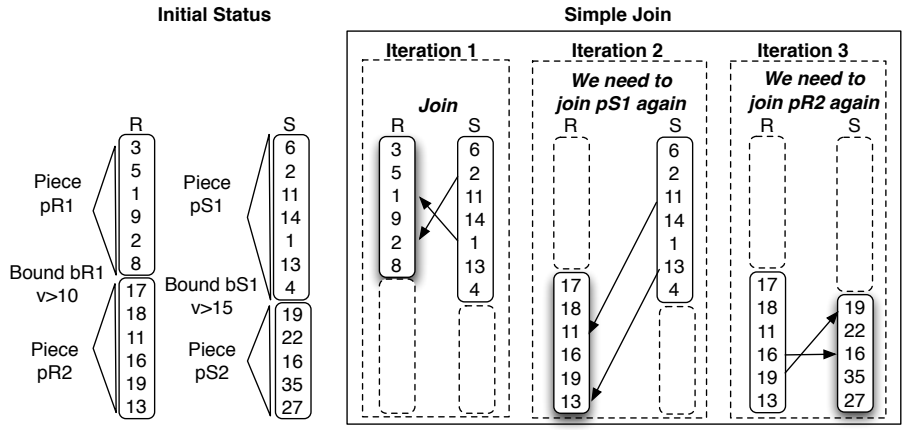


Figure 6.1: A simplified example of the simple join

6.2 The Basic Crack Joins

In this section, we discuss the problem of exploiting the knowledge gained via cracking such as to perform efficient joins. We gradually explore the possible directions starting with a simple algorithm that exploits the partitioning knowledge in the cracker columns in a straightforward way. Then, we introduce more advanced algorithms that also perform on-line physical reorganization of the input columns enhancing the performance and the self-organization properties. A detailed experimental analysis demonstrates the potential of the approach, while the introduction of the cache conscious and the active crack joins is discussed in the next sections. The introduction of the cache conscious and the active crack joins is discussed in the next sections.

6.2.1 The Algorithms

The Simple Join

The motivation for the first algorithm, the *simple join*, is that by using the information in the cracker indices of the two columns to be joined, we can identify pieces that span over similar/overlapping value ranges across the two columns. These pieces can be joined *independently* without having to consider the rest of the tuples for possible hits.

In order to exploit the partitioning knowledge, we need to consider the following issues regarding the two input columns.

- They have been cracked by *different queries* leading to pieces spanning over different value ranges for each column.
- They have been cracked by a *different number* of queries leading to a different number of pieces for each column.

The challenge is to cope with these issues and exploit any cracking knowledge to benefit during the join. The simple join takes a straightforward approach; walking through the pieces over both columns, it continuously tries to identify and join pieces with overlapping value ranges.

The procedure is described by Algorithm 8. A simplified step-by-step example is given in the left box of Figure 6.1. The algorithm performs a merge-like scan over both columns, successively joining each piece R_i of column R with all (consecutive) pieces S_j of column S whose value ranges overlap with the value range of R_i. The cracking knowledge is nicely exploited since we avoid joining pieces that are known for sure not to contain any matching values.

Performing multiple joins on smaller inputs rather than one join on larger inputs is beneficial as it improves spatial locality of data access. In particular excessive cache misses due to random access into the inner side are limited, if not avoided, once the pieces fit in the CPU caches (Shatdahl et al., 1994; Boncz et al., 1999).

Overall efficiency is additionally improved by the fact that each join of two matching pieces is executed independently, allowing crack join to make individual optimization decisions for each join. Given the properties (e.g., size and data distribution) of both pieces, it chooses at runtime not only the most suitable join algorithm (typically a hash-join unless both pieces are very small (nested loops) or happen to be sorted (merge join)), but also decides per join which piece is used as inner input. Always using the smallest piece as inner input optimizes random access locality while probing the inner side. In Figure 6.1, the inner pieces are marked by shading, e.g., the arrows that indicate the matching values point into the hash tables (assuming a hash join).

Naturally, the more queries touch/crack the join inputs during selections, the more knowledge we get and the more efficiently we can process both selections and also joins over these columns. The question is; can we do even better?

Algorithm 8 simpleJoin(ColumnLeft, ColumnRight) Perform an equi-join between ColumnLeft and ColumnRight.

```
1: left = indexLeft.firstPiece
2: right = indexRight.firstPiece

3: while true do
4:    join(left.piece, right.piece)

      {Advance to the next piece from the column with the lower bound}
5:    if left.bound < right.bound then
6:       left = left.nextPiece
7:    else if left.bound > right.bound then
8:       right = right.nextPiece
9:    else
10:      left = left.nextPiece
11:      right = right.nextPiece

12:   if left.piece == indexLeft.lastPiece or
      right.piece == indexRight.lastPiece then
13:      join(left.remainingPieces, right.remainingPieces)
14:      return
```

The Cutter Join

The simple join algorithm, described above, effectively exploits the available cracking knowledge. The benefit (i.e., performance gain) of the simple join is limited by the fact that, in general, the two join input columns have been cracked independently. Hence, the resulting pieces are *not aligned* between the columns, overlapping only partly instead of matching exactly. Consequently, each piece typically has to be considered more than once to be joined with all partly overlapping pieces of the opposite input column (e.g., see Figure 6.1). Furthermore, due to its *passive* nature, the simple join is bound to "wait" for more selections to crack the columns into more, and hence, smaller pieces to further improve the join performance.

These observations lead to the *cutter join* algorithm. Instead of merely exploiting the existing "imperfectly" cracked input columns by joining all combinations of partly overlapping pieces, the cutter join *actively* invests in physically reorganizing the join columns on-the-fly, using cracking to synchronize the piece

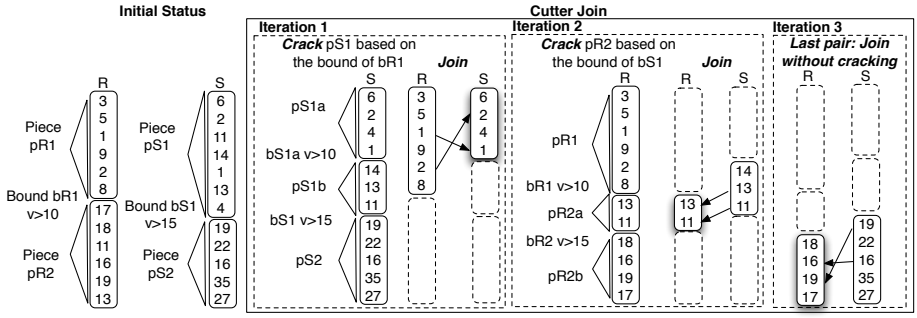


Figure 6.2: A simplified example of the cutter join

boundaries, and hence, create *aligned* pieces that pairwise cover identical value ranges. This way, more, and thus smaller, pieces are created, reducing the join costs per piece, while the total number of joins does not increase, as now each piece needs to be considered *at most once*.

The exact procedure is described in Algorithm 9 while Figure 6.2 shows a simplified example with two columns that have been previously cracked in just two pieces. Similar to the simple join, the cutter performs a synchronized scan over the pieces of both columns. However, at each iteration and before joining a pair of matching pieces, if these pieces do not cover *identical* value ranges, then the piece with the larger range is first cracked, using the bounds from the smaller range r, to create a piece that matches r. The new matching pieces are then joined and the algorithm proceeds to handle the next pair of pieces. Thus, each piece from one input matches (at most) one piece from the other input exactly, requiring (at most) one join per piece. To perform the actual cracking we use the algorithm described in (Idreos et al., 2007a). Similar to the simple join, for each individual pair of pieces, the cutter dynamically chooses the most suitable join algorithm and inner input based on the properties of the pieces. The benefit is twofold.

- The investment in cracking is considered cheaper than the anticipated reduction in join costs. This way, we expect to see an immediate benefit for the current join.
- Any cracking performed during a join is logged in the cracker indices. This way, we expect to see a benefit on any future crack operator on this column(s) by exploiting the extended cracking knowledge.

Algorithm 9 cutterJoin(ColumnLeft, ColumnRight) Perform an equi-join between ColumnLeft and ColumnRight by on-the-fly cracking non aligned pieces.

```
1: left = indexLeft.firstPiece
2: right = indexRight.firstPiece

3: while true do
4:    {Crack pieces with not an identical value range}
5:    if left.bound < right.bound then
6:       right = crack(right.piece, left.bound)
7:       if right.isEmpty then right = right.nextPiece

8:    else if left.bound > right.bound then
9:       left = crack(left.piece, right.bound)
10:      if left.isEmpty then left = left.nextPiece

11:   else
12:      {Join pieces with an identical value range}
13:      join(left.piece, right.piece)
         {If one of the columns has no more pieces}
         {then skip the rest of the other side pieces}
14:      if Ileft == IndexLeft.lastPiece then
15:         return
16:      else
17:         Ileft = Ileft.nextPiece
18:      if Iright == IndexRight.lastPiece then
19:         return
20:      else
21:         Iright = Iright.nextPiece
```

The Smart Cutter Join

Typically, DBMSs try to maximize the benefits and "over-amortize" the investments of creating auxiliary data structures by keeping them available as long as possible to be used more than once. For example, after a hash join the system can keep the created hash table and reuse it in future joins with the same inner input, saving the costs for repeating the preparation phase. In this section, we discuss this optimization issue in the context of the crack joins. We introduce a variant called the *smart cutter join* that maintains and reuses hash tables

whenever possible. There are two main issues to consider.

- The crack join does not build a single big hash table on top of one of the two input columns. Instead, it builds multiple smaller hash tables on individual pieces that may belong to either input depending on piece properties.
- The multiple hash tables are built for pieces that belong to cracker columns. These columns are continuously physically reorganized within both select and join operators. Naturally, any physical changes invalidate a hash table.

We need to cope with both these issues to correctly reuse hash tables. The first step is to properly register each hash table with the piece of the column that it was built for. We do this by extending the respective entry in the cracker index of the column with a reference to the piece's hash table.

The cutter join is modified such that in every iteration, after executing a join on a pair of pieces, R_i, S_i, where a new hash table ht was built, say for R_i, a reference to ht is marked in the cracker index entry of R_i in R. Furthermore, prior to joining each pair of pieces, R_i, S_i, the smart cutter checks whether a hash table exists for either or both of the pieces. In case only one piece has a hash table, say R_i, it is reused as inner input unless S_i is small enough to fit in the cache and R_i is not. In case both pieces have hash tables, the smaller hash table is reused as inner input.

Additionally, we avoid stale hash tables by modifying the physical reorganization algorithm of (Idreos et al., 2007a) such that it drops existing hash tables of newly cracked pieces. Our decision to simply drop the outdated hash tables instead of updating them is based on two reasons. First, cracking always happens on at most two pieces of a column at a time, hence, dropping hash tables for these pieces means only a small local loss of information. Secondly, with decreasing piece sizes via continuous cracking we can exploit continuously smaller inner input pieces while rebuilding ever smaller hash tables (only when required) is a rather inexpensive task.

Updates

Continuous cracking within select and join operators results in continuous physical reorganization of all columns involved. Thus, in order to maintain the self-organizing behavior it is essential to properly handle updates on these columns since an update naturally results in physical changes.

Update algorithms for cracking have been proposed and analyzed in detail in Chapter 4. An update is not applied immediately. Instead, it remains as a *pending* update and it is applied only when a query needs the relevant data assisting the self-organizing behavior. This way, updates are applied *while* processing queries and affect only those tuples relevant to the query at hand. For each cracker column, there exist a pending insertions and a pending deletions column. An update is merely translated into a deletion and an insertion. Updates are efficiently merged in a cracker column without destroying the knowledge of its cracker index. using the Ripple algorithm. When for example inserting a new tuple, the trick is to place it in such an area of the column as to easily maintain the information of the cracker index with the minimum necessary physical reorganization actions. This brings continual reduced data access and a self-organizing behavior after and during updates.

6.2.2 Experimental Analysis

In this section, we continue with a detailed experimental evaluation of our crack joins implementation in MonetDB. We show that they bring a significant advantage by providing both an increased performance and a self-organizing behavior. For all experiments, we use a 2.4 GHz Intel Core2 Quad CPU equipped with one 32 KB L1 cache per core, two 4 MB L2 caches, each shared by 2 cores, and 8 GB RAM. The operating system is Fedora 8 (Linux 2.6.24).

Experimental Set-up

Using two 8 byte wide columns, R and S, each experiment contains multiple jobs, and each job has three steps; (i) a random selection over R, (ii) a different random selection over S and (iii) a join on R and S. Each experiment includes a sequence of 10^2 jobs. For simplicity we will refer to jobs as queries. We also test with the standard hash join operator of MonetDB as a reference point. Each selection is on a random range and with random selectivity. All crack join variants use the crack select, while the standard hash join uses the standard scan select.

Basic Observations

For our first experiment we use two columns of 10^7 tuples each. Each column contains unique values in $[0 - 10^7)$ randomly distributed. This way, the join performed at each iteration is a one-to-one join. For this basic experiment we

also include results for a sort merge join strategy. The results are shown in Figure 6.3.

Let us first concentrate on the overall performance of the join algorithms. Figure 6.3(a) demonstrates the total join costs for each algorithm through the whole query sequence. For the hash join, we observe that the first join is slower but then all subsequent joins are faster. The reason is that the first query has to build a hash table to perform the hash join. However, the hash table is not dropped and thus all future queries can reuse it and evaluate the join faster. All crack join algorithms significantly improve over the basic hash join. With every iteration the performance improves even more providing a self-organizing behavior, i.e., the more queries touch the columns, the more the join performance improves. After roughly 30 queries, the cutter joins manage to even outperform the merge join, ignoring the high initial investment of sorting both inputs.

The improvement is a result of the different access patterns involved during the join. Even though everything (the two columns and the hash table) fits easily into main memory, the modern hardware properties and trends transfer the bottleneck into the system's cache (Boncz et al., 1999; Ailamaki et al., 1999; Manegold et al., 2002; Shatdahl et al., 1994). During a hash join, both building the hash table on the inner input and probing it with the outer input inherently perform random access patterns spread over the whole hash table. High performance can only by achieved, in case such random patterns are limited to a data area that fits into the cache. Otherwise performance suffers from extensive cache misses.

I.e., accesses to values that are physically located close together (e.g., in the same cache line) do typically not occur shortly after one another but are rather widely spread in time. In case the hash table fits entirely into the cache, each cache lines needs to be loaded only once and then stays in the cache avoiding the need to load the cache line again to serve later accesses to other co-located values. In case the hash table exceeds the cache capacity, accesses to non co-located value between two access to co-located values are likely to force the original cache line out of the cache to make room for new ones. This way, a cache line has to be re-loaded every time a value is accessed.

The standard hash join uses a single big hash table that fits into the cache only if the whole columns are smaller than the cache. On the contrary, all crack join variants join many pairs of small pieces instead of two large columns, and can choose the smaller piece of each pair to build the hash table on. The more queries arrive, the more the columns are cracked into smaller pieces. The pieces eventually fit into the cache, improving join performance significantly due to increased spatial locality of random access patterns.

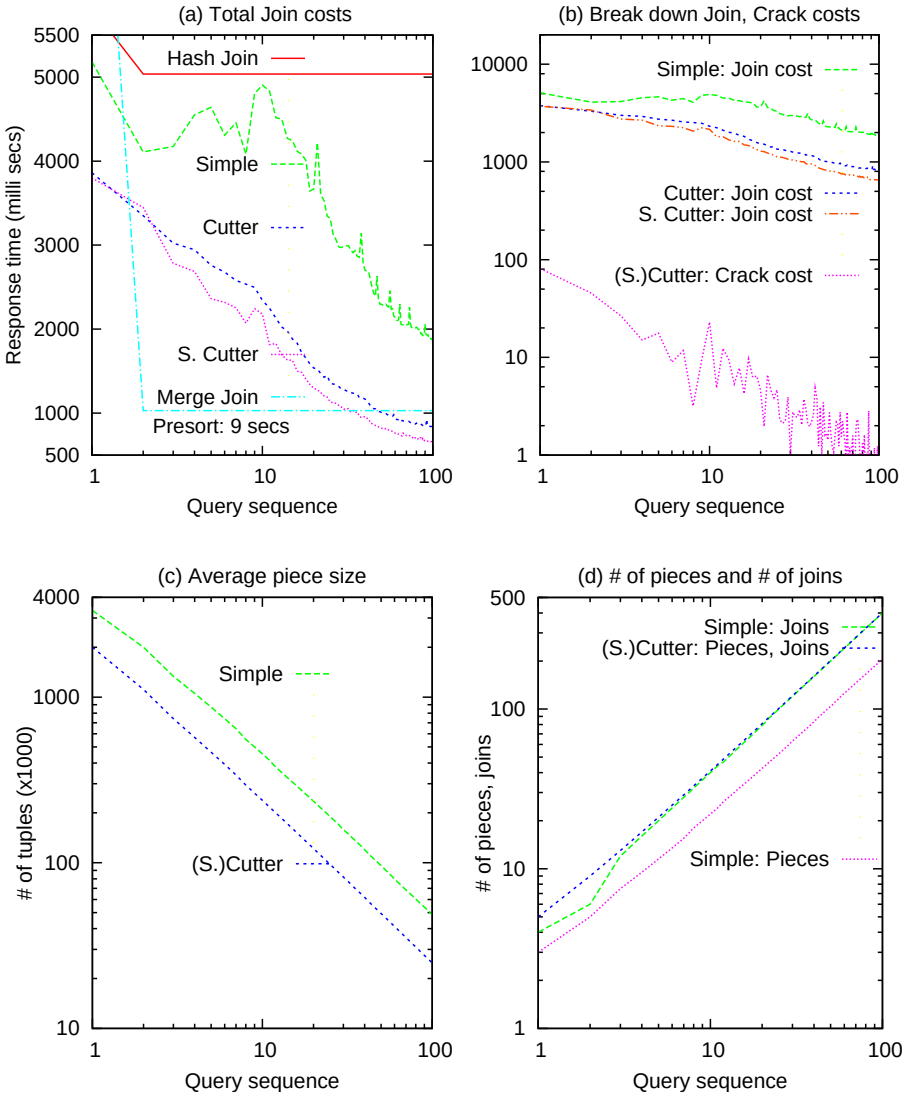


Figure 6.3: Improving join processing with crack joins (10M $\bowtie$ 10M $\Rightarrow$ 10M)

Figure 6.3(b) breaks down the total cost of the crack joins and shows for each algorithm the pure join cost, i.e., the cost spent purely in creating and probing hash tables. We see that for all crack joins, the pure join cost is much lower than what we see in Figure 6.3(a) for the standard hash join. The smart cutter materializes a bigger benefit by correctly maintaining and reusing hash tables whenever possible. In addition, Figure 6.3(c) shows how the average size of the pieces in the columns is decreased during the query sequence allowing to continuously handle smaller and smaller pieces.

Simple Vs. the Cutters

Let us now focus on the differences in the behavior between the crack join variants. Figure 6.3(a) shows that both cutter joins significantly outperform the simple join. The advantage comes from the fact that the cutters do not simply exploit the knowledge gained via crack selects as the simple join does. Instead, they perform further cracking while processing a join such that the join inputs are physically reorganized to align their pieces.

This improves the pure join cost of the cutters over the simple join as seen in Figure 6.3(b). This phenomenon is explained better in Figure 6.3(d) where we show for each algorithm the number of pieces in the columns and the number of joins needed during the whole query sequence. The cutter and the smart cutter have identical results due to having the same cracking strategy. For the cutters, the number of pieces in a column is identical to the number of joins performed as by aligning the pieces across the two columns, a cutter join can safely join each piece of a column with at most one piece of the other one. On the contrary, the simple join has to handle pieces with overlapping ranges across the two columns and thus each piece might have to be an input for more than one joins. Figure 6.3(d), shows that even though the simple join has a smaller number of pieces compared to the cutters case, it still needs to perform a similar amount of joins in order to produce a complete result.

Furthermore, as seen in Figure 6.3(c), the cutters join continuously smaller pieces than the simple join and thus they exploit better access locality. This is due to the fact that with the cutters the columns are cracked both during a join and during a selection, while with the simple join the columns are cracked only during selections. This way, every cutter join operation improves future join operations.

The overhead for the reduced join cost of the cutters over the simple join is the on-the-fly physical reorganization performed to align the pieces. Figure 6.3(b) demonstrates that this overhead (crack cost) is very small compared

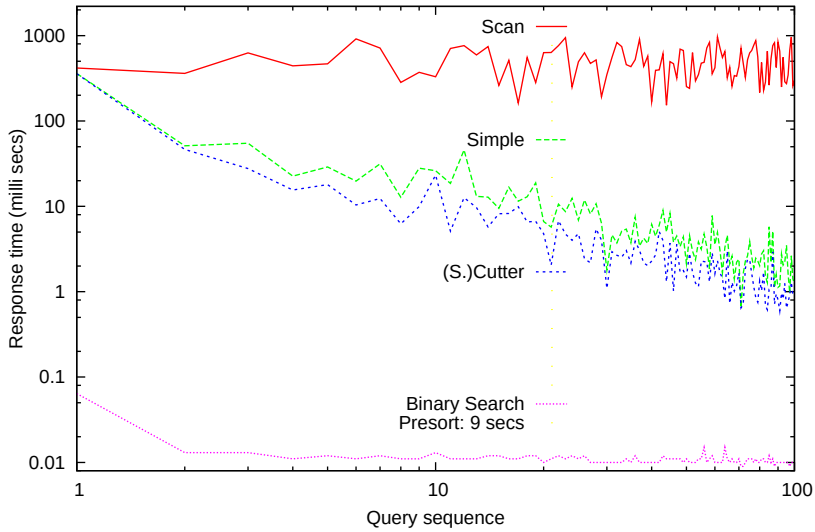


Figure 6.4: Selection costs

to the pure join cost as cracking is a very fast operation allowing to materialize a significant benefit in terms of total cost (seen in Figure 6.3(a)), i.e., the benefit of the pure cutter join cost over the pure join cost of the simple join is much larger than the overhead of cracking. Moreover, it is important to mention that the cost of cracking is continuously decreasing assisting the overall self-organizing behavior of the cutter algorithms. This happens because, the more we crack a column, the smaller pieces we create, which in turn need less effort to be physically reorganized within future queries.

Selection Improvements

In Figure 6.4, we show the total cost of each set of selection actions that interleave a join in our experiment. For all cracking cases, the more queries are involved, the faster the selection becomes as with more queries physically reorganizing the columns, the more partitioning knowledge we get. The cutters manage to improve performance even further compared to the simple join by cracking also within a join. Naturally, every cracking action performed as part of a cutter join, is subsequently logged in the cracking indices. This adds knowl-

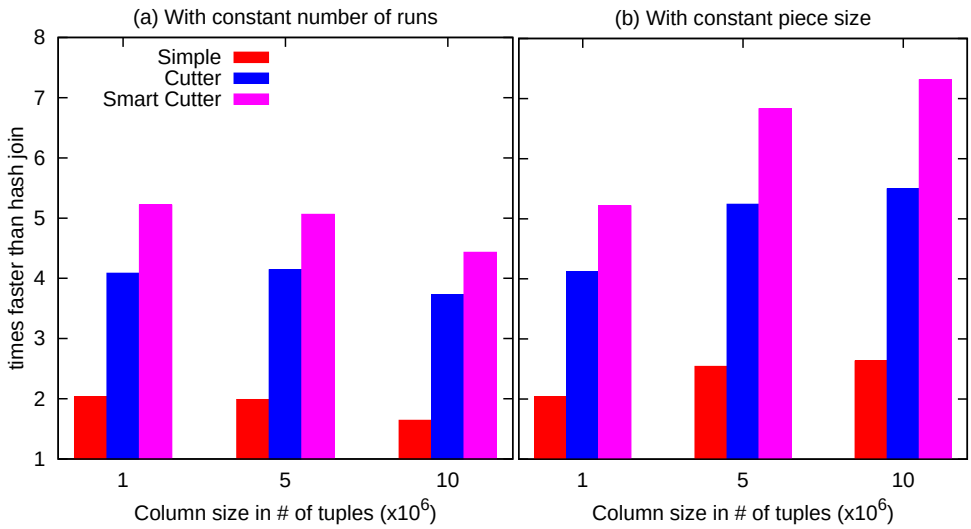


Figure 6.5: Varying input sizes

edge to the system that every future operator (not just future joins) can exploit. The benefit is more visible early in the query sequence, i.e., the cutter joins allow selections to reach the maximum performance even faster (in terms of processed queries).

Varying Column Sizes

Here we perform the same experiment as before, but this time we vary the column sizes s, with s=$\{10^6, 5*10^6, 10^7\}$. For each crack join algorithm C and each s_i, we measure the cumulative join costs for running the whole query sequence and we report the relative performance improvement, i.e., the ratio of the cumulative cost of the hash join over the cumulative cost of C.

For Figure 6.5(a) we run the same number of queries, i.e., 10^2, for all column sizes. The smart cutter provides the ultimate performance regardless of the column size. As the column sizes grow, the improvement becomes smaller for all algorithms. The reason is that within the same number of queries, a smaller column is cracked into smaller pieces compared to a larger column and thus the

crack joins can exploit more beneficial access patterns.

We repeat the experiment again for Figure 6.5(b), but this time the query sequences are adjusted to the column size in order to achieve (on average) physical pieces of similar size when we reach the last query. With columns of $x * 10^6$ tuples we use sequences of $x * 10^2$ queries. Figure 6.5(b) shows that this time, as the column sizes grow, the benefit of the crack joins over the standard hash join increases as a result of using smaller pieces in the bigger columns too. Naturally, the small pieces (same size on average in all cases) bring a bigger improvement with the bigger columns.

Various Scenarios

The previous experiments studied the case of one-to-one joins. This is a simple scenario that allows to easily grasp the basic behavior and motivation of the crack joins. The next experiment studies the behavior in a variety of scenarios using the same set-up as before with two columns R, S, but each time we vary a number of parameters, e.g., using different sizes, with different value ranges, with duplicates, etc. Figure 6.6 shows the relative per query improvement of the smart cutter over the standard hash join for the following 5 scenarios, demonstrating that it maintains a self-organizing behavior and an increased performance independently of the scenario. The cutter join exploits every opportunity to ignore pieces by exploiting the crack knowledge and by continuously cutting the columns into smaller pieces, it progressively improves even more.

a) Scenario 1. Here R contains 10^7 tuples while S is 10 times smaller and each column contains unique values in $[0 - 10^7)$ randomly distributed, i.e., not all tuples from R match a tuple in S making skipping of pieces possible. In such scenarios, the crack joins can completely skip pieces that due to their value ranges, it knows that they do not to have a matching tuple on the other side. As more queries crack the columns, the cutter has more opportunities to completely ignore pieces from R.

b) Scenario 2. To concentrate on the effect of using columns of different sizes, Scenario 2 uses the same column S as Scenario 1, but a different R column. R is created as a union of 10 S columns and then the tuples are randomly mixed, i.e., S is again 10 times smaller than R but R contains duplicates and all values in R join with a tuple in S. The cutter materializes a significant benefit and shows a self-organizing behavior by exploiting continuously smaller pieces.

c) Scenario 3. The next scenario studies the effect of skipping pieces in columns of the same size. We use the same R as in Scenario 2 but this time S is a column of 10^7 tuples containing unique values in $[0 - 10^7)$. This means that

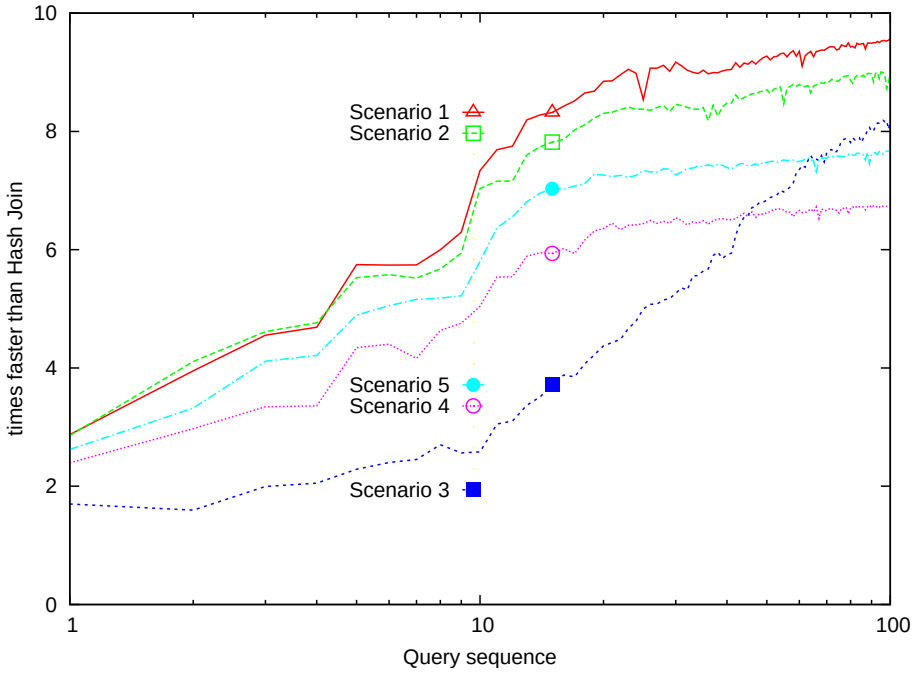


Figure 6.6: Various scenarios

10% of the tuples in S match with multiple R tuples (i.e., 10 each) but 90% of the tuples in S do not match with any R tuple allowing the cutter to skip pieces. Again the performance of the cutter continuously improves via cracking.

e) Scenario 4. Here we use two columns of 10^6 tuples each, with the same unique values in $[0 - 10^7)$ randomly mixed; we have columns of the same size both containing unique values with gaps and every tuple has a match. Again, the cutter exploits the continuously smaller pieces to improve performance although no skipping of pieces is possible.

d) Scenario 5. Here we simulate a foreign key scenario. Column R contains 10^7 tuples while column S is 10 times smaller. S contains unique values in $[0 - 10^6)$, while R contains duplicate values again in $[0 - 10^6)$, i.e., each tuple in the small column, matches with multiple tuples in the big one. As before, the cutter exploits continuous cracking to provide a self-organizing behavior.

6.3 The Cache Conscious Crack Join

The previous section introduced the crack joins and demonstrated that they can gracefully exploit the knowledge gained during cracking to perform efficient joins. Via on-line physical reorganization they provide the system with even more knowledge and manage to continuously reduce data access and improve processing in both future joins and selects.

For ease of presentation, until now our study was concentrated on the immediate benefits of the crack joins, e.g., our experiments studied quite short query sequences. Here, we show that continuous physical reorganization without any boundaries or control results in decreasing performance during long query sequences. To overcome this problem, we introduce a more advanced algorithm, the *cache conscious crack join*, that balances the various costs involved during a crack join with respect to the hardware properties, i.e., cache sizes.

6.3.1 Long Query Sequences

For an easier understanding, of the problem, and the parameters that affect the performance, we introduce the issue through an experiment. We rerun the basic experiment of Section 6.2.2 with columns of 10^6 tuples, but this time we let it run for a much longer sequence of 10^4 queries (instead of 10^2). Figure 6.7(a) shows how the total costs evolve. All crack joins initially improve performance in a self-organizing way. During roughly the first 400 queries, the more queries touch the columns, the more the performance improves due to the creation and exploitation of continuously smaller pieces. However, as the query sequence continues further, the performance worsens significantly. The more queries are processed, the more the performance decreases. In the end, the advantage of all crack joins is lost.

To acquire a better understanding of the phenomenon, Figure 6.7(b) breaks down the various cost components of the smart cutter. These are: (*i*) the join cost, i.e., the cost spent purely in joining the various pieces, (*ii*) the crack cost, i.e., the cost spent in physically reorganizing the columns and (*iii*) the administrative cost, i.e., the rest of the cost spent within the algorithm as to maintain the status at each step, call the appropriate functions, perform all kind of checks, e.g., check when we are ready to join a pair of pieces, when should we move to the next pair, update the cracking indices, check for (or invalidate) existing hash tables, etc.

Figure 6.7(b) clearly shows that the increased total cost is a result of the increased administrative cost. Figure 6.7(b) helps to assess the situation better

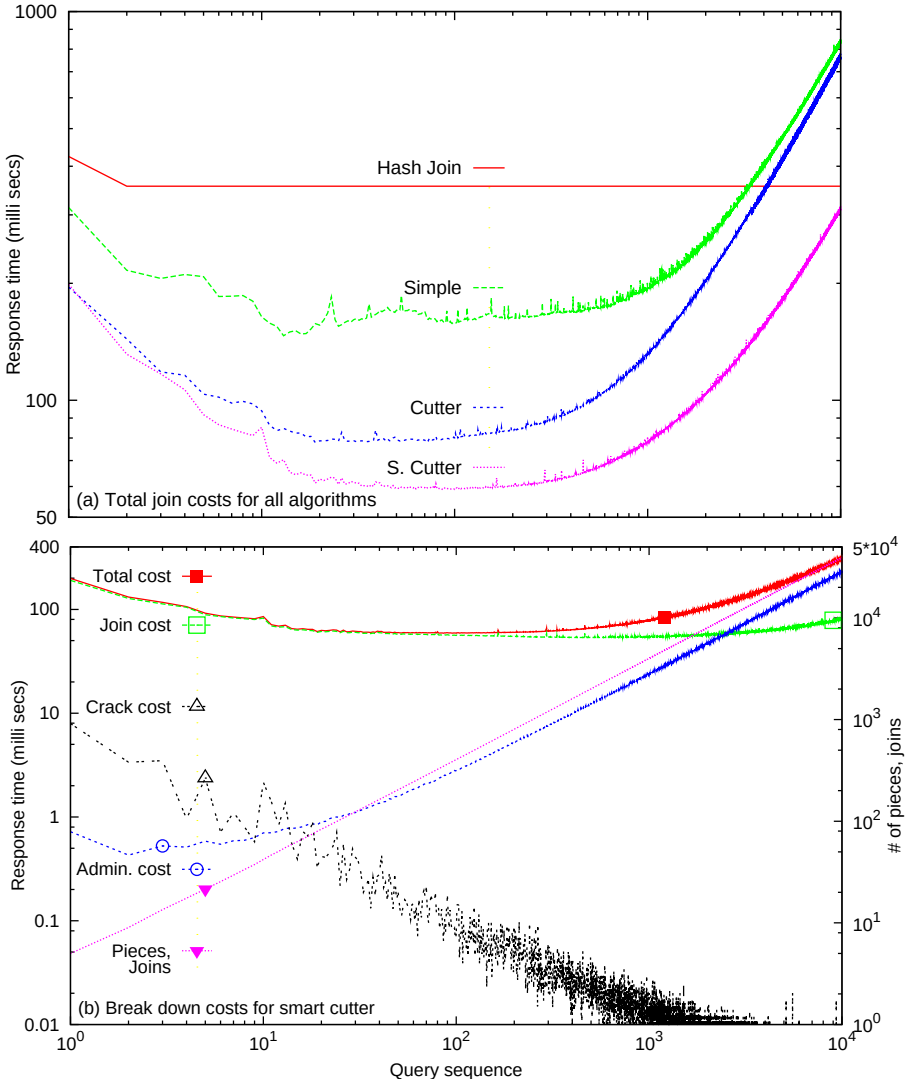


Figure 6.7: Side-effect of very long query sequences

by also depicting the number of physical pieces in each column after each query (secondary y-axis). Given that the algorithm fully aligns the two input columns, both columns contain the same number of physical pieces after each query. As discussed in Section 6.2, the number of individual joins performed by our smart cutter join algorithm is equal to the number of pieces in either column.

The administrative cost shows the same behavior as the number of pieces, i.e., the more pieces the algorithm has to handle, the more the administrative cost increases. This is natural since with more pieces, the algorithm has to perform more iterations/joins. For each iteration/join it needs to perform all the controls and checks all over again taking into account the specific status for the pair of pieces at hand.

Observing Figure 6.7(b) more carefully, shows that the pure join cost is also increased in the end of the query sequence. This is a slight increase that does not significantly affect the total cost. The analysis shows that again this is due to the involved administrative cost. The hash join is a very complicated and big piece of code and naturally, within the hash join code there is a number of function calls. The more times we call the hash join, the cumulative cost of simply calling the various functions within the join increases. In the example of Figure 6.7, this phenomenon appears after the point where each crack join performs roughly $15 * 10^3$ joins.

6.3.2 Cache Conscious Crack Join

The previous section, demonstrated the long query sequences issue and pinpointed the factors that cause this behavior. Here we use the above observations to craft a solution.

Balancing the Costs

The rising challenge is to find the proper *balance* for the cost components of the smart cutter. The benefits of the cutter joins come from significantly reducing the pure join cost by continuously cracking the columns into smaller pieces. Figure 6.7(b) clearly shows that there is a turning point, where although we keep cutting the column, we stop bringing any more benefits in terms of the total cost both because the pure join cost stops improving but more importantly because the administrative cost explodes. As seen in Figure 6.7(b), the total cost performance remains stable at its optimal level for a sequence of several queries. The goal is to *identify* this area and *maintain* this performance through the whole query sequence.

Avoid Restricting Physical Pieces

One solution could be to stop cracking a column after it reaches an "optimal" number of P pieces. Both the administrative and the pure join costs would not increase after this point and we only need to properly define P such that it reflects one of the points in the area where the total cost is optimal. However, such a solution is not beneficial. Some examples are given below.

- Crack select operators would become significantly slower since without the ability to continuous crack columns we would have to scan pieces we are not allowed to crack. Each column is independently cracked by both crack joins and crack selections. If we had to stop cracking within selections, then we would need to completely redesign the cracking selection algorithms and operators and we would dismiss the continuous cracking benefits in this area by introducing scanning of pieces that we are not allowed to crack anymore.

- Crack joins would become significantly slower as after some point we would not be able to crack a column in order to achieve piece alignment across two join inputs. As the columns are continuously cracked by both joins and selections, this would mean that after some point the crack join would have to operate on non-aligned columns and thus incur a decreased performance.

- A given column can participate in joins with multiple other columns, each one cracked in a different way and thus requiring a different alignment. A threshold in the number of pieces again leads to non-aligned joins.

The above examples are just a few of the reasons why going for a piece threshold over the columns is not beneficial. In the rest of this section, we present our solution that maintains the optimal performance of the smart cutter through long query sequences without imposing any threshold in the number of physical pieces in a column.

Using Super Pieces

The decrease in performance comes from the rising administrative cost needed to do a large number of cheap joins which in turn is needed because of a large number of tiny pieces. In order to reduce the number of joins, we introduce the notion of *super pieces*. A super piece is a collection of consecutive physical pieces and we will put a threshold on the possible super piece *size*. The cache

conscious join operates similarly to the smart cutter but instead of joining one pair of physical pieces at a time, it joins one pair of super pieces at a time.

We postpone for a while the issue of how large or small a super piece should be and first concentrate on how we need to adapt the steps of the smart cutter assuming super pieces of size X. At each iteration, the new algorithm first creates one super piece from each input column as follows. The procedure is simple; starting from the one side, it collects enough physical pieces until the total size of the batch is as "close" as possible to X. It continuously takes the next physical piece and once the total super piece size S becomes bigger than X, the last piece is kept only if $S - X < X - Sp$, where Sp is the total super piece size without considering the last piece. Note, that the above procedure is a logical one, i.e., no copying is involved.

Getting super pieces of exact size X is possible only if the collection of physical pieces happens to be of this exact size. Otherwise, we need to crack the pieces. However, given that we have no knowledge on which bound we should use to achieve the size X, we need to analyze the data distribution within the pieces leading in a significant overhead. Therefore, we choose a configuration that is as close as possible to X given the existing physical pieces each time.

Once the super piece of the one side is created, we proceed to create the super piece for the other side such that the last physical pieces of the two super pieces have overlapping value ranges. This is necessary in order to avoid joining a super piece (or part of a super piece) more than once.

The next step is to align the super pieces if necessary. Using the bound of the last physical piece in each super piece, we crack if necessary one of the two columns to fully align the super pieces. Then the algorithm proceeds to join the two super pieces. The hash table reuse procedures also need to change as now the hash tables are built for a collection of physical pieces at a time. This reflects mainly a technical challenge. Now each hash table is associated with *all* relevant physical pieces and it is invalidated if *any* of these pieces physically changes, i.e., it is cracked or updated during any crack operator.

6.3.3 Tuning the Super Piece Size

Using super pieces effectively puts a threshold on the number of smaller joins the crack join needs to do (relative to the size of the column). The open issue is how to determine the optimal size of the super pieces.

In this section, we study the issue of deciding the proper super piece size. For ease of presentation and understanding of the various parameters, that affect the decision we use the following experiment. Using our basic experimental set-up

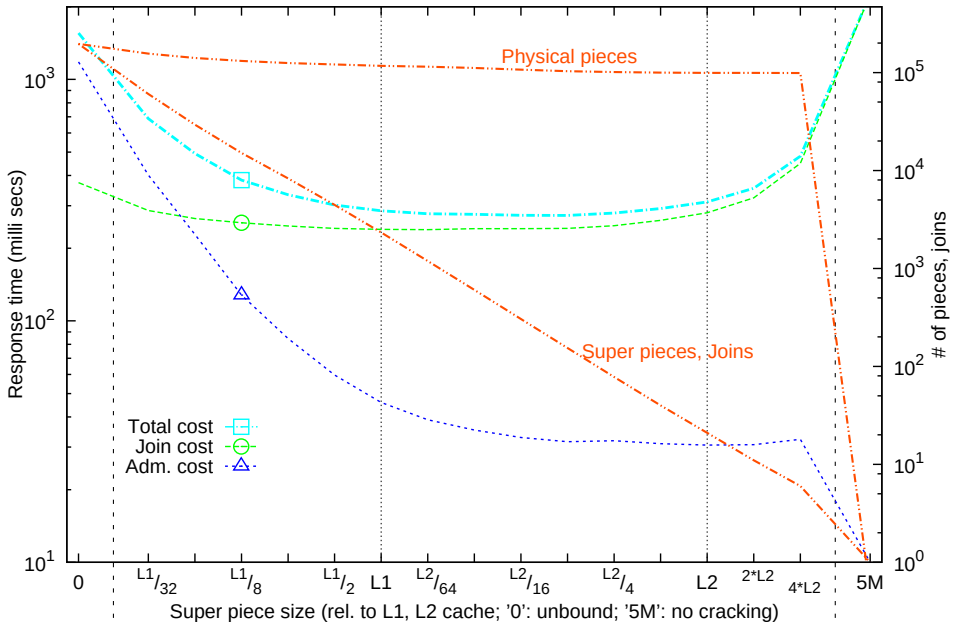


Figure 6.8: Effect of super piece size (5M ⋈ 5M ⇒ 5M)

as in Section 6.2.2, we perform long query sequences of $5{*}10^4$ runs on columns of $5{*}10^6$ tuples. Each run is repeated for various super piece sizes from well within the L1 cache to well out of the L2 cache. Each column tuple is 8 bytes while in the used machine the L1 cache is 32 KB and the L2 cache is 4MB.

Figure 6.8 shows for each super piece size the performance of the last query q in the query sequence, i.e., query $5 * 10^4$. In particular, the left y-axis reflects the total cost to run the join operator for q and breaks it down to the respective administrative and pure join costs. The right y-axis depicts the number of super pieces used to perform the join of q and the number of physical pieces per column after q is finished. The point 0 on the x-axis reflects the unbound case of joining individual physical pieces as the plain smart cutter algorithm does. The last point on the x-axis, marked as 5M, reflects the other extreme of using a single super piece spanning the whole column (5 Million tuples), which essentially means performing a standard hash join over the entire inputs. The figure clearly shows that we get the optimal performance when the super pieces

fit within the L2 cache. Let us discuss the various trends in more detail.

In the unbound case, we need to perform as many joins as the number of physical pieces, which in turn leads to an explosion of the administrative cost. However, as the super piece size increases, the performance continuously improves, materializing a significant benefit compared to the unbound case. Larger super pieces require fewer joins causing less administrative overhead.

The pure join cost increases with the unbound super piece size. As we discussed earlier, this is due to the internal administrative costs involved. The unbound case needs to perform $2 * 10^5$ joins within q. However, as we increase the super piece size, the pure join cost quickly improves since less joins need to be performed. Once the super piece size exceeds L2 cache capacity, the join cost increases significantly due to excessive cache misses as discussed in Section 6.2.2. Since the access costs between L1 and L2 differ less than between L2 and main memory, a similar increase in join cost is hardly visible when the super piece size exceeds L1 capacity.

The number of physical pieces also decreases slightly as the super piece size grows. This is because within the join, cracking happens only in order to align the edges of super pieces. This leads to less cracking (for the same initial column) as the super piece size grows. When the super piece size becomes equal to the column size, there is only one piece and we need to perform only a single but very expensive join. Given that data has to go through L1 anyway, the ultimate pattern is to make sure that while performing a join in each iteration of the algorithm, both the inner super piece at hand and its hash table fit comfortably in L1. In addition, there should be enough room for the tuples of the outer super piece although in these case we do not necessarily need to fit all of it within L1 as we need each tuple only once to perform the respective prompt. This way choosing a super piece size of half the size of L1 is a safe choice.

Figure 6.8 demonstrates that using super pieces that fit into the L2 cache yields the most benefit as it achieves the desired balance between the pure join cost and the administrative cost. This phenomenon is also demonstrated in Figure 6.9 where we run the same experiment as before, but now varying also the size of the columns from $2 * 10^6$ to $64 * 10^6$ tuples. The query sequences are adjusted to the column size in order to achieve (on average) physical pieces of similar size when we reach the last query. With columns of $x * 10^6$ tuples we use sequences of $x * 10^4$ queries. Figure 6.9 reveals that, for all column sizes, the best performance is achieved with super piece sizes that fit well within the L2 cache.

Each curve in Figure 6.9 is additionally marked to reveal even more useful information. For example, an empty upside down triangle shows the point where

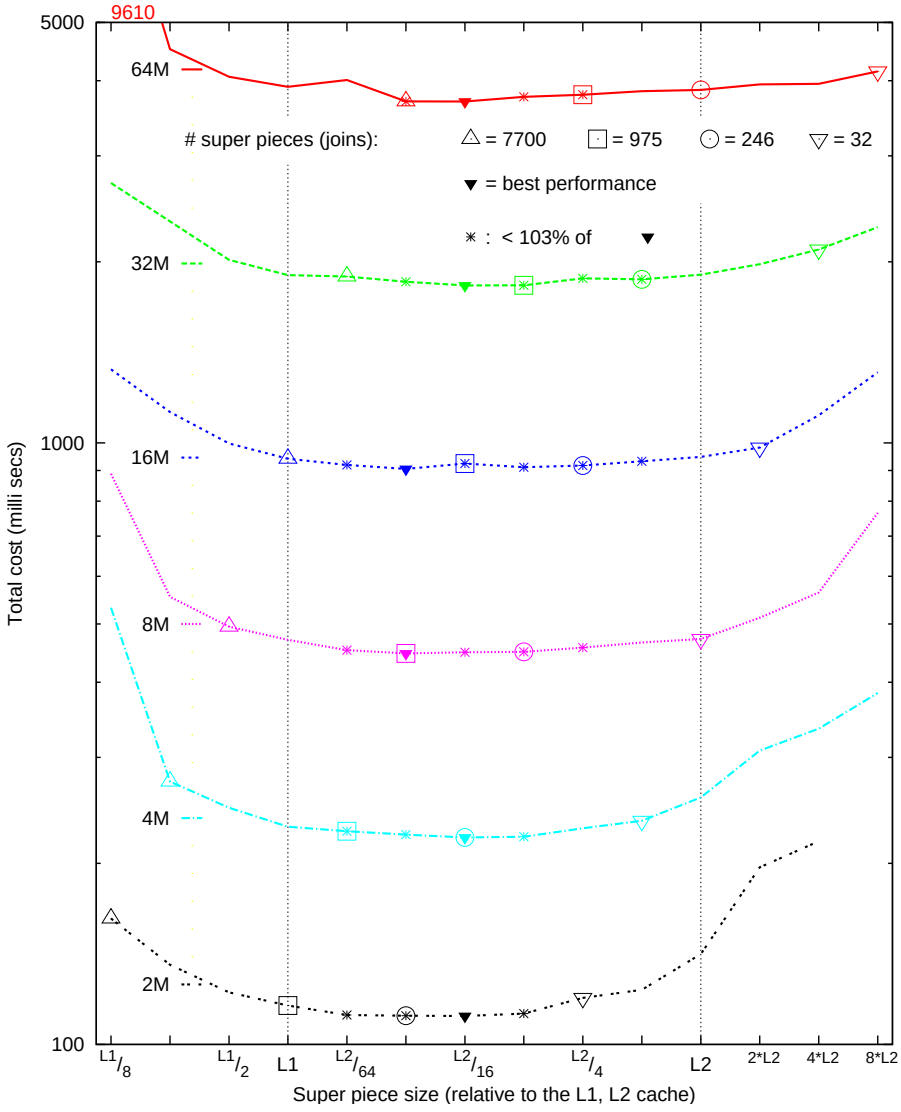


Figure 6.9: Varying the super piece and column sizes

32 super pieces are being used, a circle the point where 246 super pieces are used, etc. Similarly, each asterix shows the points in each curve where performance is just 3% off from the optimal one which in turn is marked with a filled upside down triangle.

While the optimal super piece size is "somewhere" between 1/32-*nd* and 1/16-*th* of L2 capacity, varying the super piece size between, say, 1/32-*nd* and 1/4-*th* of L2 capacity (in some cases even beyond) changes the performance by less than 3% with all column sizes. This means that a close-to-optimal super piece size can be chosen from this range, independently of the column size. An other observation from Figure 6.9 supports this statement. With larger column sizes we can "afford" to handle more super pieces (and hence joins) than with smaller column sizes. Drawing a virtual line to connect all similar marks that indicate super piece numbers reveals this trend. For example, connecting all circles (i.e., 246 super piece marks) shows that although for the smallest column size (2M) this is a number that gives a performance close to the optimal one, for the largest column size (64M) this number results in a performance much worse than the optimal. For 64M columns we need at least 1000 super pieces to get performance close to the optimal. The reason is that the administrative cost grows only with the number of pieces, but not with the column size. The join cost, however, grows with the column size, "hiding" higher administrative costs the larger the columns (and hence join costs) are. Obviously, for columns that are smaller than the observed range of close-to-optimal super piece sizes, the whole column is considered a single super piece.

Finally, Figure 6.10 depicts the per query costs during a long query sequence using columns of $4 * 10^6$ tuples. The unbound case shows the same performance degradation beyond 10^3 queries as in Figure 6.7. A rather small super piece size of $L1/8$ reduces this degradation, but does not avoid it completely. Likewise for a super piece size that is twice as big as L2. An optimal super piece size of $L2/16$ effectively avoids the explosion of the administrative cost, and hence, not only avoids the performance degradation, but even continues to show performance improvements till the end of the query sequence.

6.4 The Active Crack Join

The crack joins exploit and even enhance the cracking knowledge introduced by crack selections providing an improved performance and a self-organizing behavior. But what happens if no previous query performed a crack select on the current join inputs? Here, we discuss this issue in detail and we present

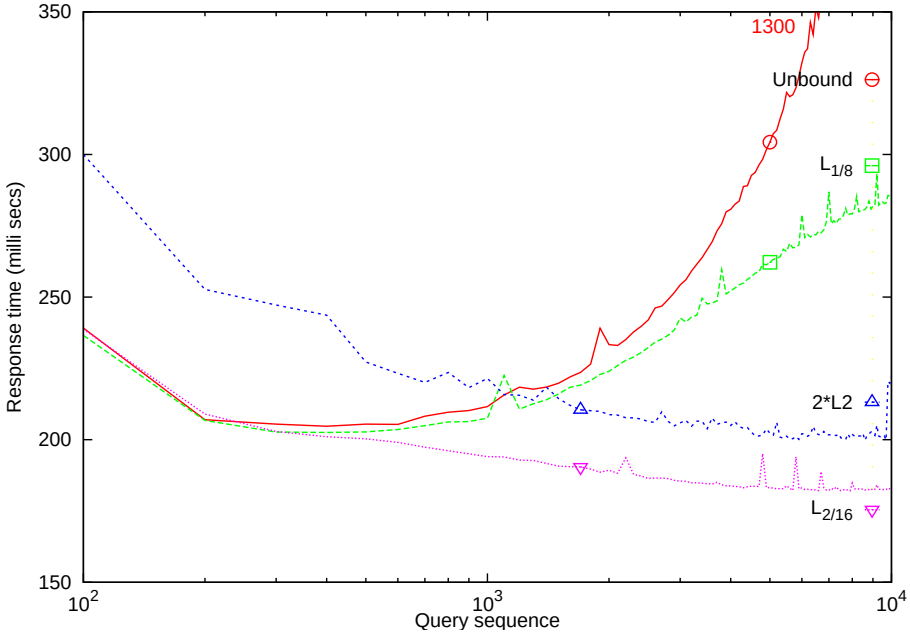


Figure 6.10: Per query costs during long sequences

the *active* crack join that introduces cracking on-the-fly without the need for an external trigger.

6.4.1 Basic Observations

In this section, we discuss the problem in detail and make the basic observations towards a solution.

Passive Cracking

Assume a join between columns R and S. The crack joins purely exploit the knowledge introduced by selection operations of any previous query on R and S. Even the cutter variations, perform cracking only for alignment reasons which is again a side-effect of a previous select operation, i.e., R will be cracked for alignment during a crack join only because such a crack operation has been

previously performed on S in a crack selection. In this sense, the crack joins have a *passive* behavior, waiting for crack selections on either input to introduce more cracking knowledge.

Exploit Alignment

The first observation is that there is no degradation compared to the basic non-cracking performance. On-line alignment is a powerful technique that helps to minimize the problem cases. For example, assume that *only one* of the join inputs, say R, has been cracked by selections in the past. In this case, by exploiting on-the-fly alignment, the crack join simply transfers all cracking knowledge from R to S, allowing the join to fully exploit the properties and benefits of crack joins.

The real problem case is when *both* R and S have not been cracked in the past. Then, there is no cracking knowledge to exploit or transfer via alignment. The crack join treats then the whole column of each input as a single piece. However, this is the same as what happens in a non-cracking column-store. Thus, there is no degradation in performance. The question is if we can do better than that and exploit the full power of crack joins in more cases.

Problem Generalization

In fact, we can generalize the problem at the granularity of individual *pieces* in a cracker column, i.e., pieces that are *big enough* so that we could benefit from cracking them, but they are not further cracked because no crack select in the current workload reorganizes these value range areas in either join input. A completely uncracked column is simply an extreme case of this problem.

More Crack Operators

Other than selections (Idreos et al., 2007a; Idreos et al., 2007b), tuple reconstructions (Idreos et al., 2009) and now joins, we expect the introduction of many more crack operators. For example, aggregations or group-by operators are natural candidates that can both take advantage of existing cracking and also perform on-the-fly cracking on their inputs, e.g., a group-by can crack a column based on the needed groups. Thus, crack joins are expected to benefit from (and also assist) the actions of a much larger number of operators. We leave this discussion for future work as it is orthogonal to this chapter.

6.4.2 Active Cracking

The problem comes from the *passive* nature of the crack joins, i.e., they wait until another crack operator introduces cracking in the join inputs, while cracking within a join is used only for alignment. In this section, we introduce the *active* crack join. The characteristic of the active join is that in addition to alignment cracking, it also performs on-the-fly cracking in order to *reduce* piece sizes. It is built on top of the cache conscious join of Section 6.3 and differs in the following ways.

Candidate Pieces

The active join on-the-fly identifies pieces that are candidates for further splitting. These are pieces that are bigger than the super piece size used to avoid excess administrative costs as seen in Section 6.3. There is no need to invest resources into cracking pieces smaller than the super piece size as these are ignored, anyway. This way, *before* joining two pieces, these pieces are first cracked to further reduce their size if necessary. Once two pieces need no more crack operations, either for active cracking or for alignment issues, only then we proceed to join these pieces and then the algorithm moves to the next pair of pieces.

Splitting

To actively crack a piece p of a column C, the high and low value bounds of p in the cracker index are used, i.e., the minimum and the maximum possible values in p (see Fig.6.1 for an index example). If no crack operator touched C in the past, i.e., no index is available, the bounds for the initial active crack are taken from the statistics of C. A number of splitting alternatives are exploited.

- *Half:* The value range is cut in half and p is cracked into two new pieces. If one of the new pieces is empty, e.g., because of a skewed distribution, then p is recracked using the bounds of the non-empty piece. The process is repeated until two new non-empty pieces are created (if possible).

- *Sample:* This is similar to the previous one but we change the cracking algorithm so that it autonomously chooses the splitting bound by performing small (within the cache) sampling steps to the edges of the piece. The crack algorithm starts reading anyway from both edges, so this incurs a small overhead at the benefit of a more informative bound.

- *Random Draws:* Next we exploit random bound draws with the expectation that they should quickly converge to the splitting goal at a smaller overhead. The *random half* technique repeatedly draws random bounds and cracks p in two pieces while the *random range* draws a random range within low and high and cracks p in three pieces in one go. The intuition is that cutting in three pieces should converge more quickly to the goal in more skewed distributions.

Strategies

We consider multiple strategies regarding when active cracking is applied on a candidate piece;

a) Single Column. This strategy performs active cracking only on the biggest candidate piece of each join input. Thus, in every join, at most two active cracks may happen, one on each input.

b) Single Piece. The second strategy performs active cracking on *all* candidate pieces the crack join finds on its way.

c) Exhaustive Piece. The final strategy, in addition to actively cracking all candidate pieces, also cracks each piece recursively until it is *not* a candidate piece anymore. It carefully prioritizes the actions in order to exploit cache conscious access patterns by always making sure that it actively cracks the most recently touched piece.

With all the above strategies, the active join manages to overcome the lack of cracking knowledge by being able to on-the-fly introduce new cracking knowledge in the columns. In the experiments section, we analyze the performance of the various strategies in more detail.

Multi-cracking and Radix-partitioning

The exhaustive strategy leads to even more interesting directions. Its side-effect is that it creates in one go multiple pieces without any workload input. It achieves that, via repeatedly cracking a column until the target size per piece is reached. Two questions arise: How does this compare with existing partitioning techniques? Can we perform all necessary cracking in one step?

In (Boncz et al., 1999; Manegold et al., 2002), we designed the Radix-clustered partitioned hash join. Radix is a state of the art cache conscious join algorithm, that in one go partitions the join inputs into 2^B partitions using the B left most bits of the attribute hash values. With large B it avoids cache-

and TLB-thrashing by performing multiple passes to limit the number of active output cursors.

The main advantage of the family of adaptive joins presented in this paper is their adaptive nature, designed to fit seamlessly in an adaptive kernel. In other words, an adaptive join exploits and assists numerous past and future operators in any query on similar attributes, while radix and similar strategies are focused on improving a single instance of a single operator call. For example, the adaptive joins can exploit past selections, tuple reconstructions, joins and with their actions improve future operators of this kind. At the same time, they maintain their high performance under updates and given that they are based on range-partitioning they can potentially be used for range joins as well.

Multi-Crack

The question is then, what can adaptive joins learn from existing one-step partitioning algorithms. Here, we design an one-step cracking algorithm that creates N cracking pieces in one go. This *multi-cracking* action is to be used the first time a column is touched for a join. The critical observation is that we do not need to actually achieve the perfect partitioning. As we have seen in previous experiments, performance for adaptive joins improves very fast so a "good enough" partitioning is sufficient. The idea is that after multi-cracking, the active join is responsible to selectively fine-tune any areas in a cracker column that might need further splitting. Not strictly requiring perfectly equal partition sizes, we achieve a "best effort" range partitioning into 2^k partitions as follows. Given the smallest ($\underline{v}$) and largest ($\overline{v}$) value in the column, by performing a single radix-sort step on the "0-aligned" values (i.e., $v - \underline{v}$) on the k most significant bits of $\overline{v} - \underline{v}$, i.e., the result partition of value v is determined by those k bits of $v - \underline{v}$ that match the positions of the k most significant bits of the "0-aligned" largest value. Investing in an extra initial scan over the column to count the actual bucket sizes, we are able to create single continuous range-partitioned result array in a single pass. Both initial counting and actual partitioning are very lean branch-free routines, consisting only of simple bit operations (and an addition for counting). Memory access to both the input column and each partition of the output columns is always sequential.

Sorting Crack Pieces

With the advent of the exhaustive piece strategy, further alternative directions arise. Once a piece in a cracker column is small to fit in the caches, we can *sort*

it. Sorting within the cache is expected to be a fast action due to the increased data locality.

With two sorted pieces from each join input, we can now exploit merge joins. As before data locality is exploited as everything happens one pair of pieces at a time, i.e., each area in the join inputs is first cracked until small enough, then sorted and then joined before going to the next area.

Sorting pieces in a cracked column has of course further benefits and side-effects, i.e., it helps with concurrency control when multiple queries need access to crack the same area of a column and it helps reducing the size of the table of contents we need to maintain for a given column. On the contrary, it increases the overhead of updates. We do not discuss further the above issues as they are general optimization issues orthogonal to this paper. Here, we concentrate on the effect of sorting to the join operator.

6.4.3 Foreign key Vs. Arbitrary Joins

In join processing we can in general distinguish between arbitrary joins and foreign key joins. Depending on the application scenario one or the other may appear more often in query plans. All our discussion so far naturally applies to arbitrary joins. Regarding foreign key joins for cracking, the discussion of the previous section is relevant. Typically there are no selections or any other operator used on top of the key or foreign key attributes. Thus the active crack join may be used. However, for foreign key joins we can do better than that by exploiting the properties of this class of joins, i.e., we know that every tuple from the foreign key side matches a tuple in the primary key side. This knowledge when combined with the ideas in this chapter and those in Section 5 allows us to craft a much more flexible and efficient solution that completely replaces the need to perform foreign key joins with cracking. We postpone further discussion on this topic for future work consideration.

6.4.4 Experimental Analysis

In this section, we present a detailed experimental analysis using the same set-up as in Section 6.2.2. We compare all active join variations against the basic hash join, the sort merge join, and the Radix-clustered partitioned hash join (Boncz et al., 1999; Manegold et al., 2002). The latter is a state of the art cache conscious join algorithm, that partitions the join inputs into 2^B partitions using the B left most bits of the attribute hash values. With large B it avoids cache- and TLB-thrashing by performing multiple passes to limit the number of active

output cursors. We always report the results of the most efficient bits/passes combination for two targets: (1) minimal total cost for clustering plus joining; (2) minimal join costs at the expense of more expensive finer clustering. For the super piece size in the active joins, we use 1/16-*th* of L2 capacity, which proved to be in the area of optimal performance in Section 6.3. Unless mentioned otherwise, we use the half splitting strategy.

Basic Performance

For our first experiment, we use two columns of 10^7 tuples each with unique values in $[0 - 10^7)$ randomly distributed. A sequence of 100 join operators are fired with no selections being involved. Figure 6.11(a) shows that even with no selections in the loop, all active joins easily outperform the hash join and manage to provide a self-organizing behavior by cracking the inputs on-the-fly, purely for reducing piece sizes. The exhaustive piece strategy provides the best behavior by quickly reaching the best performance, while the other two active join variations need more queries to reach the same performance levels.

Figures 6.11(b), (c) and (d) provide more insight on this behavior. They break down the costs depicting the pure join costs, the crack costs and the number of joins performed for each strategy. As seen in Figure 6.11(b), the pure join cost improves very quickly for the exhaustive piece strategy, while it improves much slower (in terms of processed queries) for the others. Figure 6.11(c) shows that the exhaustive piece makes a higher investment in cracking with the first query, breaking up the column into more pieces more quickly (see Figure 6.11(d)). Given that the pure join cost is the dominant cost factor, this helps to materialize a significant benefit. The other two strategies are more conservative in their cracking investments which results in a slower improvement pace. The single piece strategy outperforms the even more conservative single column strategy; it cracks for every candidate piece instead of once per column.

6.4.5 Crack Join Vs Radix and Merge Join

After the first run, the ultimate performance of crack, radix and merge join is quite similar improving significantly over the plain hash join. The differences though are in the actual properties of each algorithm. Radix was designed to improve over the merge join by having a lightweight preparation phase exploiting modern hardware properties and reaching similar performance with merge join for the joining phase. Compared to cracking though radix is slightly slower mainly because it does not allow to maintain hash tables but the crucial differ-

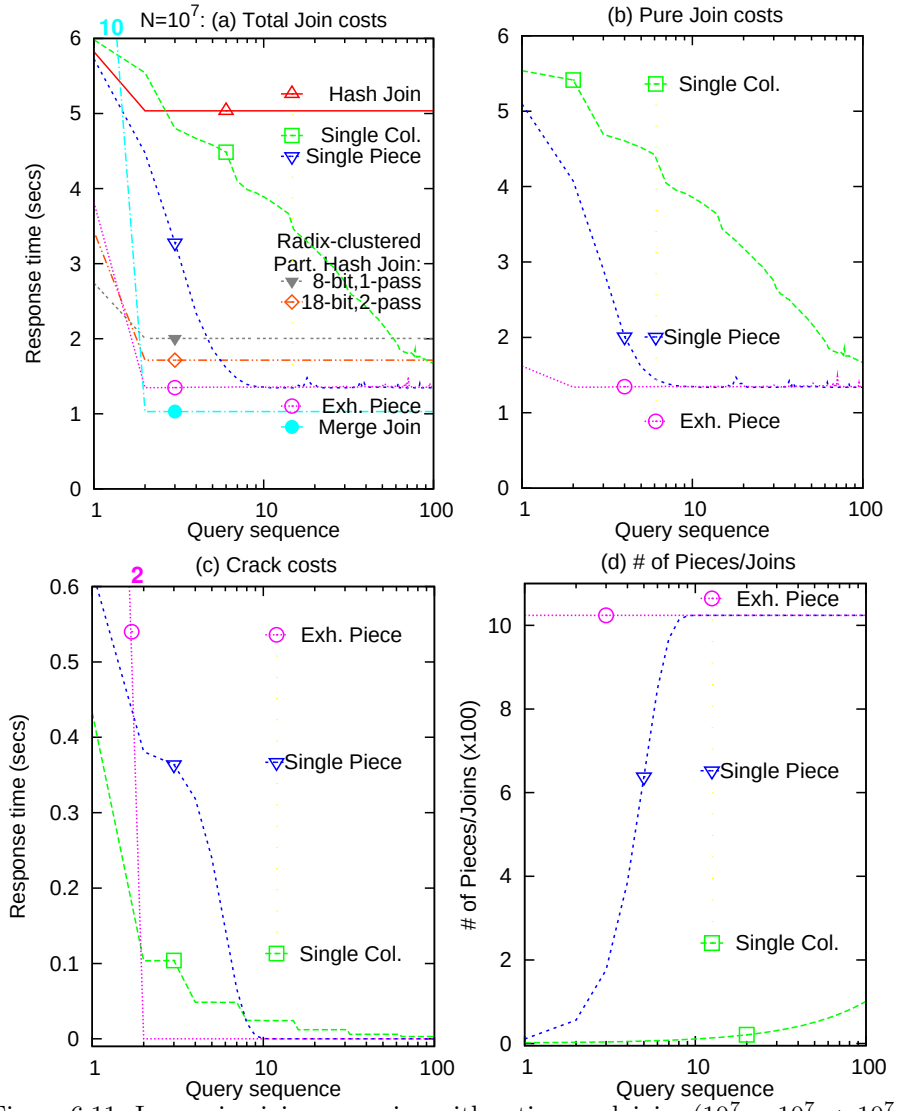


Figure 6.11: Improving join processing with active crack joins ($10^7 \bowtie 10^7 \Rightarrow 10^7$)

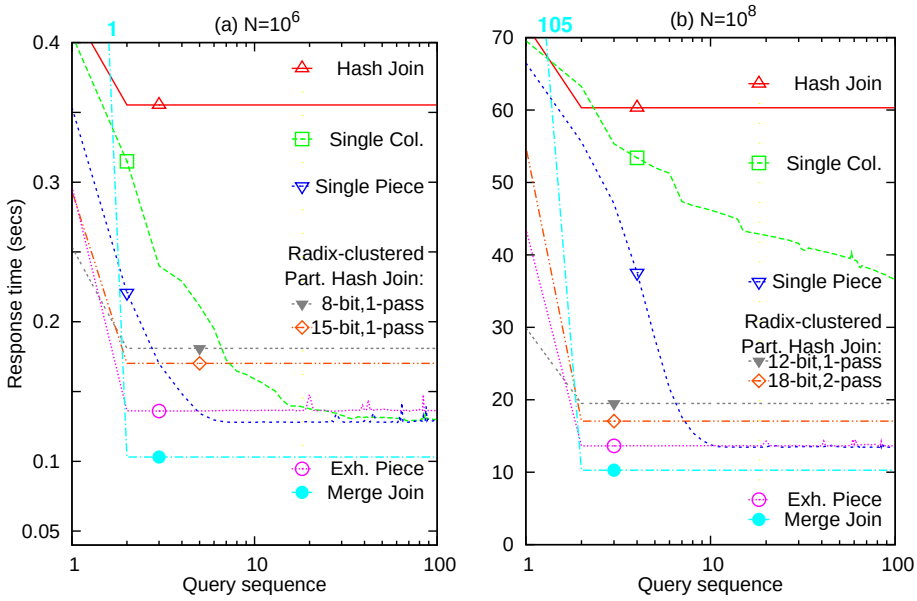


Figure 6.12: Improving join processing with active crack joins ($N \bowtie N \Rightarrow N$)

ence is that radix cluster is non-updateable (cf. Fig. 6.15) and suits only join processing (in fact only equi-joins).

When comparing cracking against merge join, we observe that merge join has a steep initial cost but then its ultimate performance is slightly better than that of crack join. This means that, in the long run, merge join will eventually gain over cracking in terms of cumulative query cost. However, merge join requires a very expensive up-front sorting step, and thus it needs (a) a priori workload knowledge and (b) idle time for the investment. This becomes more crucial as column sizes grow (cf. Fig. 6.12.b). Even if this kind of luxury is available, merge join additionally requires a stable workload so that the investment in sorting can pay off. In addition, it requires a workload with no or very infrequent updates; so far there is no efficient way of maintaining sorted columns (Harizopoulos et al., 2006).

On the contrary, cracking overcomes all these limitations of merge and radix join as it is designed for dynamic workloads. It has a low on-the-fly start-up cost which allows for instant and automatic adaptation to workload changes.

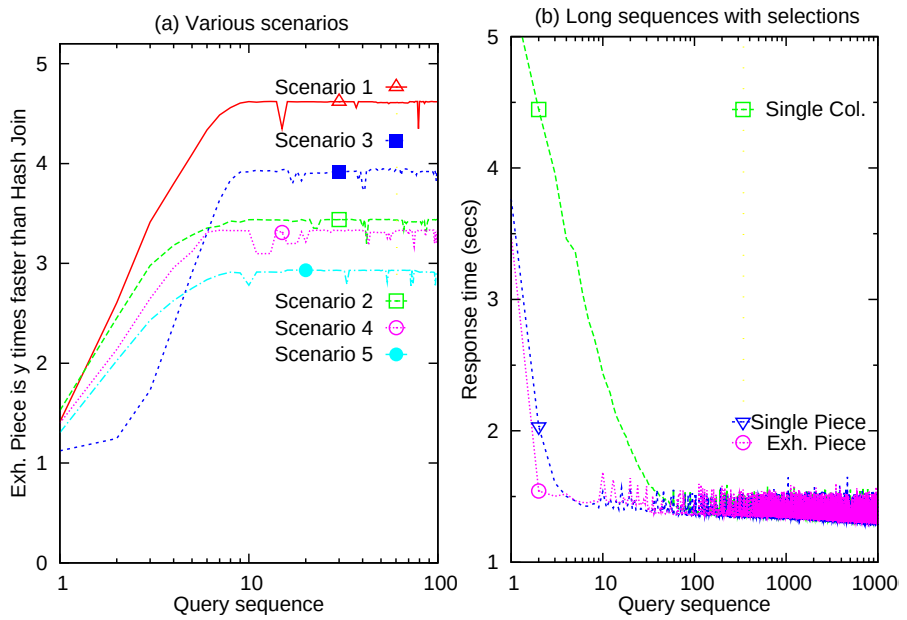


Figure 6.13: Improving join processing with active crack joins ($N \bowtie N \Rightarrow N$)

Furthermore, it is updateable (Idreos et al., 2007b) (cf. Fig. 6.15), and improves in a self-organizing way not only joins but also selections (Idreos et al., 2007a)(cf. Fig. 6.4), tuple reconstruction (Idreos et al., 2009), and potentially more operators, e.g., aggregation, group by, etc.

Varying Column Sizes

Here, we study the effect of input sizes. We perform the same experiment as before, but this time we use inputs of 10^6 tuples (Figure 6.12(a)) and inputs of 10^8 tuples (Figure 6.12(b)). The behavior remains the same as seen before, scaling nicely with input sizes. In both cases, the exhaustive piece strategy provides the best performance very quickly for the active join, while depending on the input sizes, the rest of the active join strategies need more or less queries to reach the same performance levels as naturally with bigger pieces, it takes more effort (queries) to bring the pieces to the optimal size.

Various Scenarios

Next, we study the active join performance for all 5 scenarios studied in Section 6.2.2. In all 5 scenarios, using inputs of different sizes, with duplicates or gaps in the value ranges, etc., we test a wide range of the possibilities of the active join to properly cut a column into multiple pieces. Figure 6.13(a) shows the improvement of the exhaustive piece over the standard hash join. In all cases, the active join materializes a significant benefit by being able to adapt and provide a self-* behavior to all environments.

Long Sequences

Using the same inputs as in the basic experiment, Figure 6.13(b) shows the join performance during long query sequences where random selections interleave with join operators. All active joins maintain the behavior seen in the previous graphs; they are not affected by the long query sequence and the ever increasing number of pieces (that the selections introduce) in the columns by efficiently exploiting super piece usage (as seen in Section 6.3).

Varying Skew

Here, we test the alternative splitting strategies for various degrees of skew in the column values. To create a skewed distribution of degree say 90%, 90% of the column values take a value from the 10% of the domain while the rest of the values take a value from the rest of the domain. We compare all splitting alternatives for the exhaustive cracking. Figure 6.14(a) shows the total join costs for the first query, i.e., the columns are cracked all the way while no prior selection has touched them. The first query is sufficient as all the crack cost goes there. For all joins, the higher the degree of skew, the higher the join costs. This is natural since with higher skew, we have more duplicates and the probes in the hash tables have to follow longer chains to get the proper values. Figure 6.14(b) shows how the pure join costs increase with the skew. However, all crack joins maintain a significant advantage over the hash join regardless the skew by minimizing the pure join costs.

Comparing the alternative crack strategies, we see that they all achieve comparable overall performance. Relying on efficient splitting implementation and cache conscious patterns in always recracking the most recently read piece, all strategies quickly converge in finding good splitting points.

Observing the pure crack costs in Figure 6.14(c), shows that the sample approach can have a cheaper crack cost by taking more informative bounds.

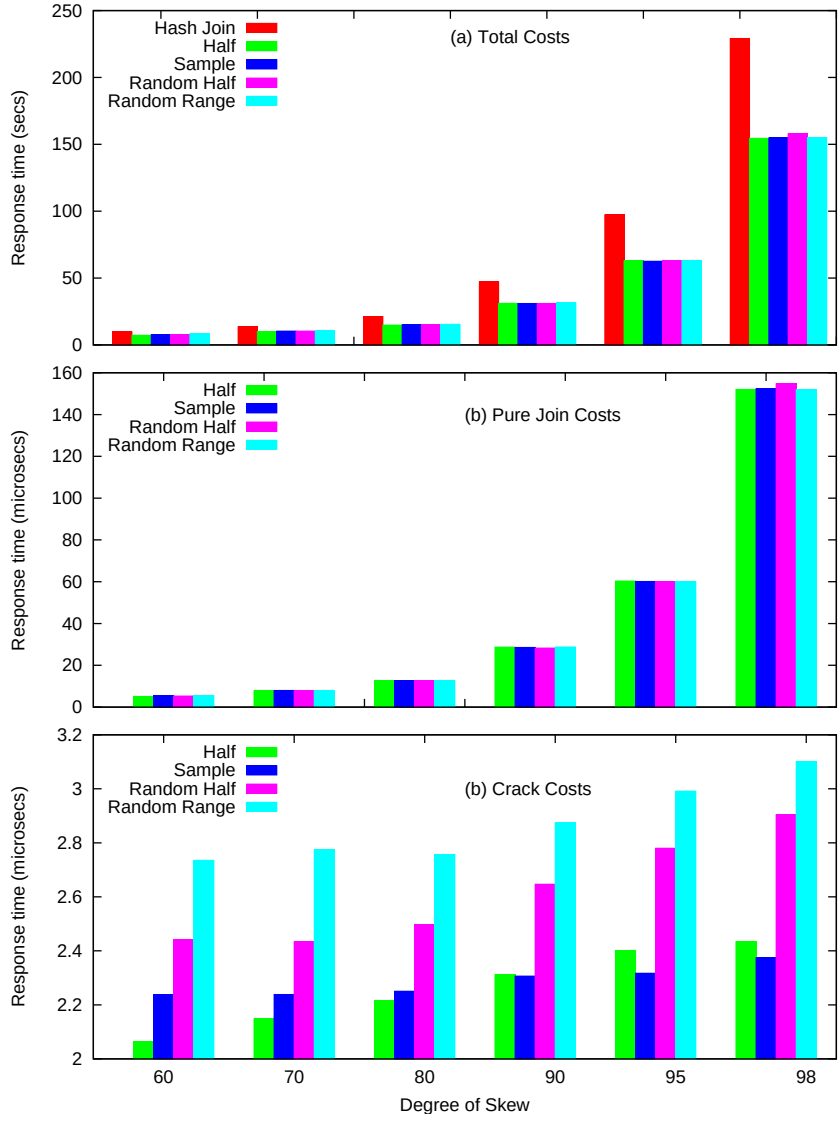


Figure 6.14: Varying skew distribution ($10^7 \bowtie 10^7 \Rightarrow 10^7$)

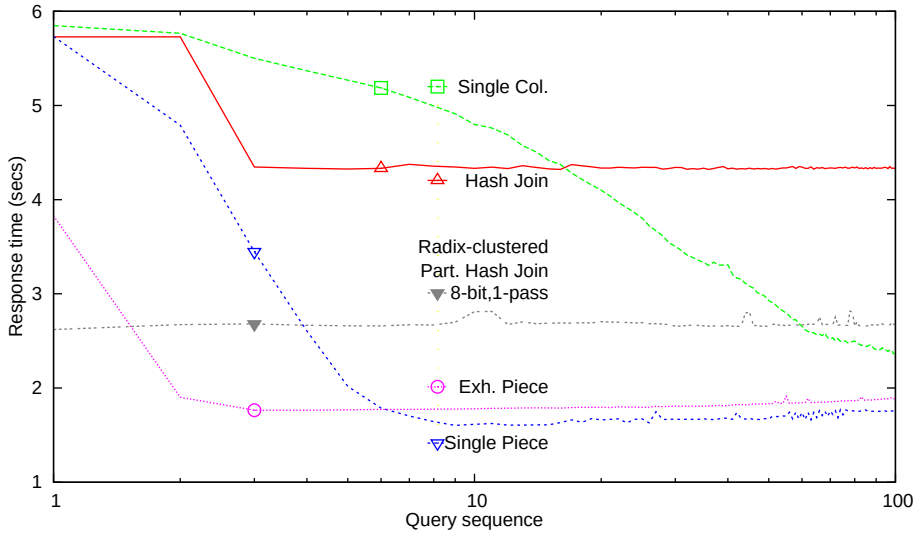


Figure 6.15: Updates ($10^7 \bowtie 10^7 \Rightarrow 10^7$)

Naturally, the higher the degree of skew, the more sampling pays off, while for very low skew, it mainly reflects an overhead compared to simply cutting the range in half. The random choices need more cracking effort as leading both in good but also in very bad choices, they can often lead to completely recracking a piece multiple times. The random range strategy has a slight overhead due to the much more complex algorithm to achieve the splitting of a piece in three new pieces in one go.

Even with the above differences, all techniques provide the same overall performance since the cost is dominated by the pure join costs. They all eventually find good splitting bounds and cut the columns in such pieces, such that the joins can be executed efficiently, materializing a significant benefit over the hash join even for the first query.

Updates

Using the basic set-up, as in our first experiment, for 10^7 columns, this time updates arrive after *every* query, inserting 10^2 random values at a time in each join input. Figure 6.15 shows that all crack joins successfully maintain their

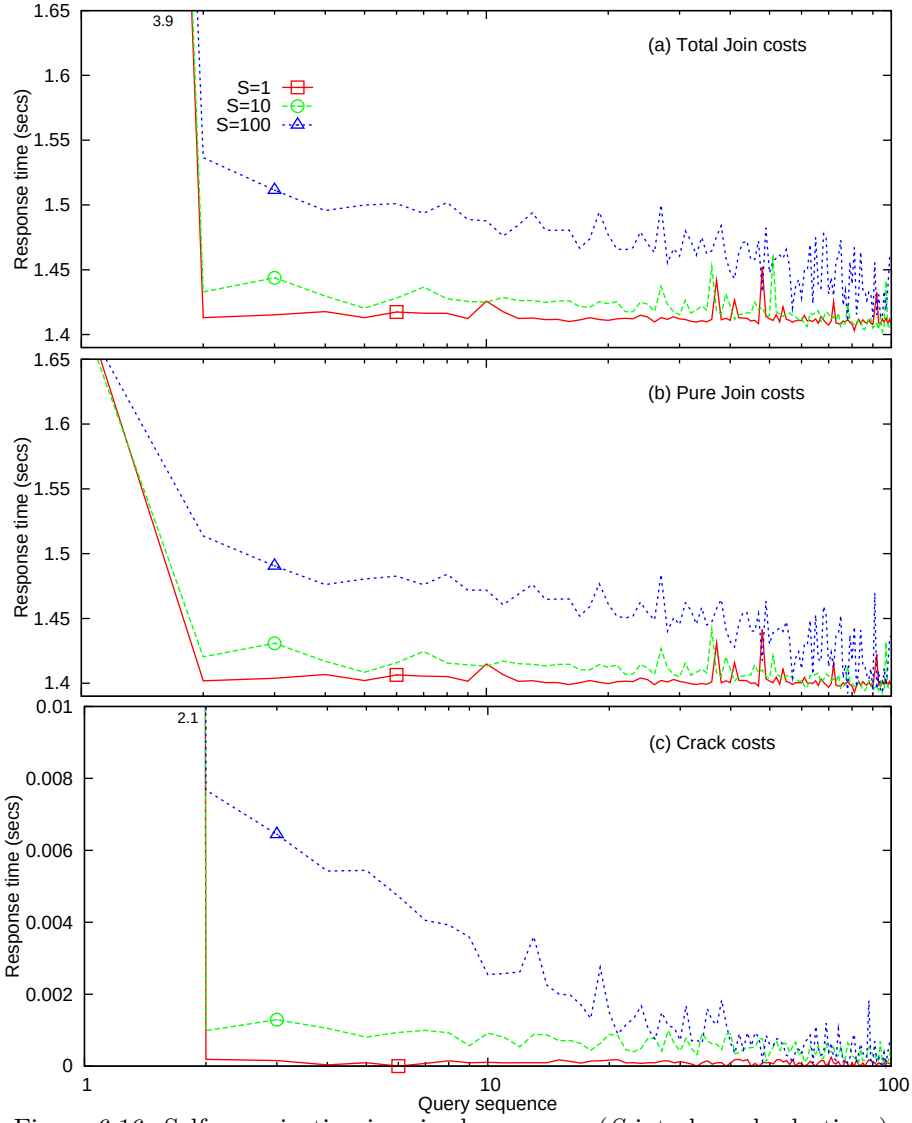


Figure 6.16: Self-organization in mixed sequences (S interleaved selections)

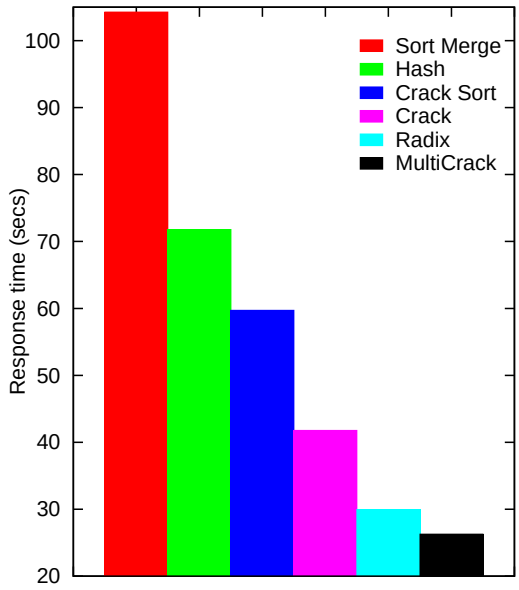


Figure 6.17: Sorting and Multi-cracking

self-organizing behavior and high performance by merging updates on-the-fly without destroying the cracking knowledge as in (Idreos et al., 2007b). For the simple hash join, MonetDB efficiently extends the hash table built by the first query so that all future queries can exploit it. Radix join on the other hand, needs to continuously repartition the inputs after every update. We do not run with merge join, since to the best of our knowledge, so far there is no efficient way to maintain sorted columns under frequent updates in column-stores (Harizopoulos et al., 2006). Obviously, resorting with every update is prohibitive.

Sorting and Multi-cracking

Here we test the exhaustive strategy using the sorting and the multi-cracking variations. Our input columns are of 10^8 values. To focus on the up-front investment, we show only the first join in the query sequence that reflects a workload change and bares the most overhead.

Figure 6.17 shows that sorting crack pieces smaller than cache size makes the first query 20 seconds slower (60s with *CrackSort* vs. 40s with plain *Crack*). The analysis shows that the pure join cost improves by 3 seconds (10s for CrackSort instead of 13s for Crack) when exploiting already sorted areas in crack columns as opposed to performing hash joins. However, this is not enough to overcome a 23 seconds penalty of sorting. The interesting part is if we compare this variation of the active join with the pure sort-merge join. It is roughly half the cost of the sort-merge join and results in exactly the same result, i.e., leaving back two fully sorted columns. This is again due to exploiting better access patterns during sorting and joining. Finally, we observe that the multi-crack variation significantly outperforms the original active join by performing most of the necessary cracking in one go, instead of repeatedly cracking the column. Its gains come purely by the effort spent in reorganizing the columns, which is 12.5 seconds for the multi-crack case, while it is 28 seconds for the initial active join. Here the multi-crack procedure creates in one go 4096 pieces and then the original active join code takes care of fine-tuning this by creating roughly another 2000 pieces while traversing the columns for the join. It even outperforms original radix, although as we discussed earlier too, the benefits of the crack join come more from its behavior over a sequence of queries and its ability to help and exploit other future and past operator depending on the workload.

Mixed Sequences

All experiments so far studied sequences where subsequent joins where interleaved with maximum one random selection on each input. Here, we study more complex scenarios and show that the self-organizing behavior of the crack joins nicely adapts to any scenario. Intuitively, if many crack operations, either selections or joins on overlapping join inputs, interleave subsequent joins, then an extra alignment effort is needed each time we rejoin the same inputs. We first concentrate on the scenario of multiple selections interleaving joins, since this is much easier to study as we know that each selection will cause a maximum of two crack operations on a column. We use our basic set-up with the exhaustive join, but this time each join is interleaved with $S = \{1, 10, 100\}$ random selections on each input.

Figure 6.16(a) shows the total join costs. For S=1 we see the same behavior as in previous graphs (e.g., Fig.6.11(a)). For a larger S however, we see a different behavior; towards the beginning of the query sequence, the cost is slightly higher (still significantly better than other approaches though, e.g., compare to Fig.6.11(a)) and progressively it drops to similar levels as for S=1.

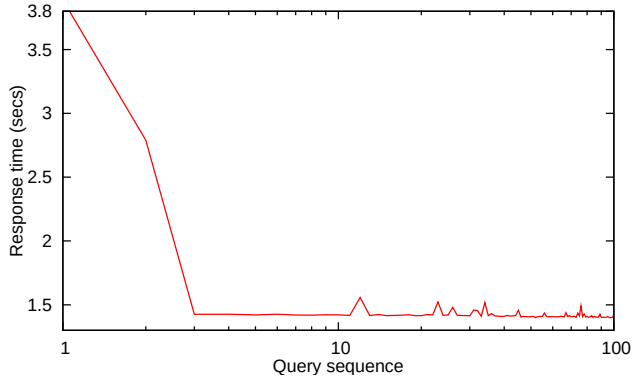


Figure 6.18: Self-organization in mixed join pairs sequences ($10^7 \bowtie 10^7 \Rightarrow 10^7$)

The effect is stronger as S becomes larger. First, observe that even with this slight extra cost the performance of the crack join is still significantly better than that of other approaches (e.g., compare with Figure 6.11(a)). Second, this behavior represents a clear self-organizing behavior with the crack join nicely adapting to the needs of the workload.

Figures 6.16(b) and (c) help to understand this behavior. With more selections in between, a larger number of cracks happen on each join input. When we need to execute a join again, the more fine-grained pieces demand a different alignment. As seen in Figure 6.16(c), this leads to higher cracking costs within the join in order to align the join inputs. However, as the pieces become more and more fine-grained, less effort is needed both because less alignment is necessary and also because pieces become smaller and thus easier to crack.

Figure 6.16(b) shows that a similar trend exists for the pure join costs too. The reason here is that the cracks in intermediate selections destroy the hash tables on any cracked pieces. Thus, subsequent joins have to build new hash tables for the new pieces. However, as pieces get smaller, less pieces need to be cracked during selections, due to the more fine-grained partitioning knowledge, leading to less hash tables being dropped. In addition, with continuously less alignment happening (Fig 6.16(c)) again less hash tables on super pieces need to be dropped. Thus, progressively the crack join adapts to the environment achieving great performance.

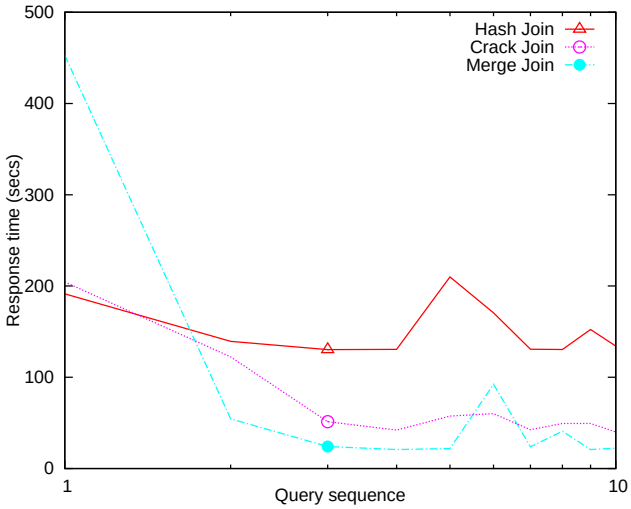


Figure 6.19: Beyond the Memory Bounds

Mixed Join Pairs

The second relevant experiment, demonstrates the effect when overlapping join pairs interleave execution. Here, each join between columns R, S_1 is interleaved with 10 joins on columns R, S_2 and each join on R, S_2 is also interleaved with 10 random selections on each one of its inputs. Figure 6.18 shows the total join costs. First, observe that different join pairs can nicely exploit all crack operations performed from a different join. For example, the first query performs the exhaustive crack join on R, S_1. This gives us the same behavior as in Figure 6.11(a). Then, the second join, on R, S_2 this time, can exploit all crack information gained from the previous join and executes much faster, i.e., it needs only to align S_2 to R. From there on, 10 random selections interleave every R, S_2 join and every 10 joins we perform a R, S_1 join again. Due to the extra alignment needed to get S_1 aligned with the more fine-grained knowledge in R we see small spikes roughly every 10 queries. However, these get smaller and smaller due to the continuously more fine-grained information as discussed in the previous experiment too. The even smaller spikes observed for R, S_2 are mainly due to the selections effect discussed before.

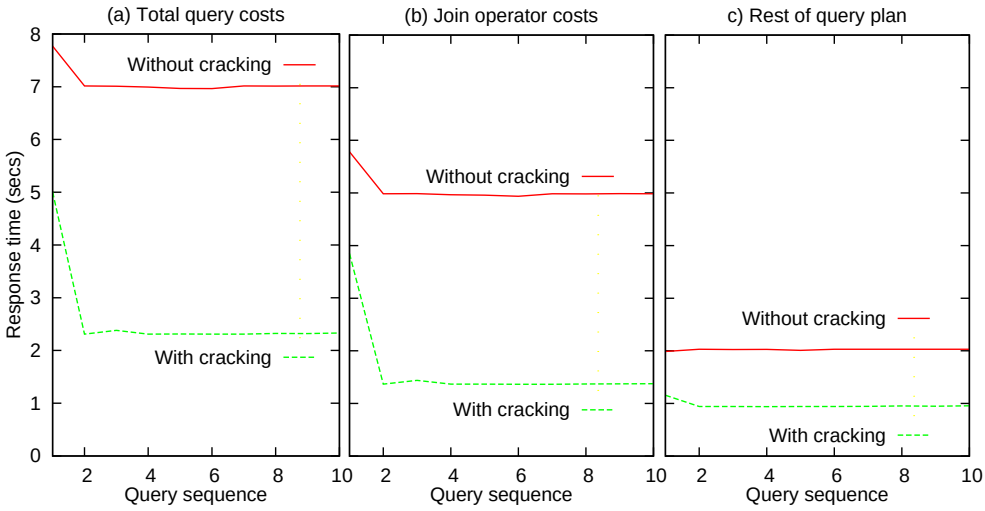


Figure 6.20: Full query performance ($10^7 \bowtie 10^7 \Rightarrow 10^7$)

Beyond the Memory Bounds.

In our next experiment, we demonstrate the performance as the data grows outside the memory limits. Here the join inputs are columns of $2 * 10^8$ tuples. The figure on the left demonstrates the performance. When compared to the behavior seen with smaller sizes, e.g., Figures 6.11(a), 6.12(a) and 6.12(b), we see that the self-organizing performance nicely scales. Crack join boosts the performance by a factor of 2 compared to the hash join while merge join achieves a comparable performance but at a very steep initial cost for the sorting step (ups and downs for all cases are due to caching and swapping effects). For cracking there is room for significant optimization by further improving the first query where crack join gets now similar performance to the hash join. Observe that for smaller sizes crack join was faster even for the first query which does all the heavy work of creating and reorganizing the cracker columns. This path calls for optimizing the basic reorganization algorithms of Chapter 3 (see also discussion in Chapter 8).

Complete Queries

So far we studied the join operator in isolation. This experiment demonstrates that the crack joins nicely fit in the big picture of cracking query plans materializing significant benefits in terms of total query cost. We use the following query over two tables of 10^7 tuples each.

```
select max(R3),max(R2),min(S3),min(S2)
from R,S where R1=S1
```

The query plan contains a single join operator, then 4 tuple reconstruction operators to retrieve the qualifying values of each necessary column and 4 aggregation operators. Figure 6.20(a) shows that when MonetDB uses a cracking query plan gets a three times better performance. Figure 6.20(b) shows separately the cost of the join operator where the gains of cracking are similar to what we observed in previous graphs.

In addition, Figure 6.20(c) shows the rest of the query plan cost, i.e., the cost to perform the tuple reconstructions and the aggregations. Cracking benefits here as well by minimizing the random read patterns during the tuple reconstruction actions. Crack joins are developed over sideways cracking (Idreos et al., 2009) which is designed to improve tuple reconstruction using cracking to *align* columns. This brings relevant values physically closer across multiple columns minimizing the tuple reconstruction costs, e.g., here the cracking performed in $R1$ and $S1$ during the join is dynamically transferred into the rest of the columns, $R2$, $R3$, $S2$ and $S3$, during the tuple reconstruction calls. Crack joins nicely fit in these query plans extending the self-organizing behavior and benefits of cracking.

Finally, observe that due to the extra administrative costs for active cracking and super piece management, the active join is slightly slower than the ultimate performance of the smart cutter, e.g., compare Fig.6.11(a) with Fig.6.3(a); a small price to pay for all the benefits seen in Section 6.3 and in this one.

6.5 Summary

This chapter studied the problem of how to exploit the knowledge gained via cracking in order to improve join processing. We present the crack join algorithms that not only efficiently exploit but also enhance the cracking knowledge by physically reorganizing the join inputs on-the-fly whenever this is necessary and/or beneficial. This way, a crack join also speeds-up any kind of subsequent

crack operators, i.e., joins, selects and tuple reconstructions, even across different queries. It maintains optimal performance during long query sequences with continuous reorganization, updates and changing workloads. A detailed experimental analysis demonstrates that crack joins provide a self-organizing behavior and bring significant performance benefits for a variety of scenarios.

Future work on crack joins aims on further improving the special class of foreign key joins (see discussion in Chapter 8). Furthermore, while focusing on equi-joins in this chapter, the fact that crack joins use range-based partitioning, as opposed to hash-based partitioning, opens new opportunities to also support joins with range predicates efficiently.

The next chapter studies adaptive merging, a technique inspired by database cracking, and discusses how we can combine ideas from both techniques to design new and more flexible indexing directions.

Chapter 7

Adaptive Indexing Hybrids*

7.1 Introduction

Inspired by database cracking, researchers from HP Labs, Goetz Graefe and Harumi Kuno, have recently invented *adaptive merging* (Graefe and Kuno, 2010a; Graefe and Kuno, 2010b). They adopt the continuous low-level reorganization and incremental index building ideas from cracking and the motivation is mainly adaptation with block-access devices such as disks, in addition to main memory. The principal goal for designing adaptive merging is to reduce the number of queries required to converge to a fully-optimized index, and the principal mechanism is to employ variable memory and CPU effort in each query. It is designed to fully exploit the time spent in I/O, and thus performs an early partial sort optimized for block-access devices. However, this comes at the cost of additional CPU overhead per query that becomes significant when input fits within memory. Partial results are then incrementally and adaptively merged in the target index as part of query processing. Contrary to cracking, adaptive merging focuses on providing a very fast adaptation process reaching quickly the final optimized stage.

*The material in this chapter has been the basis for a paper submitted for publication entitled "Adaptive Indexing" (Idreos et al., 2010c).

7.1.1 Contributions

In this chapter, we provide the first detailed comparison between adaptive merging and database cracking based on a complete implementation in a full kernel. We study the various trends and tradeoffs that occur and we propose a new *hybrid* adaptive indexing approach designed for lower overhead per query. By adopting from database cracking the design goal of minimizing per query overhead while at the same time exploiting the concept of runs and merging from adaptive merging, it combines benefits of both approaches. The net effect is that the new algorithm achieves such a light-weight adaptation that reflects a zero overhead over a full scan approach, essentially opening the road towards tuning-free systems that always adapt as the default action.

7.1.2 Outline

The rest of the chapter is organized as follows. Section 7.2 provides the necessary background describing in more detail adaptive merging. Section 7.3 then presents the hybrid algorithm and Section 7.4 provides a detailed experimental analysis. Finally, Section 7.5 concludes the paper.

7.2 Adaptive Merging

While database cracking functions as an incremental quicksort, with each query resulting in at most one sort step, adaptive merging functions as an incremental external merge sort. Under adaptive merging, the first query to use a given column in a predicate produces sorted runs and each subsequent query upon that same column performs at most one additional merge step. Each merge step only affects those key ranges that are relevant to actual queries, leaving records in all other key ranges in their initial places. This merge logic takes place as a side effect of query execution performed purely within query operators as in database cracking.

Adaptive merging enables control over the amount of CPU and memory invested into index refinement; the more resources invested, the fewer queries needed for index conversion. Judicious memory allocation can thus control run size, comparison count per record within each query, and thus overall CPU effort. However, the CPU effort required in order to achieve initial runs of substantial size is significantly higher than that required by database cracking for processing any query. That is to say, run generation imposes a substantial penalty in terms of CPU effort on this first query, although given today's CPUs,

the principal cost may be in movement in the memory hierarchy, e.g., disk I/O or cache faults.

7.2.1 Motivation for Hybrid Designs

One concern about database cracking is that at most two new partition boundaries per query means that the technique requires thousands of queries to converge on an index for the focus range. One concern about adaptive merging is that the technique requires the first query to pay a significant cost for generating initial runs. The difference in reorganization performance, i.e., the number of queries required to have a key range fully optimized, is due to merging with a high fan-in rather than partitioning with a low fan-out of two or three and to merging a query's entire key range rather than only dividing the two partitions with the query's boundary keys. The difference in the cost of the first query is primarily due to the cost of sorting the initial runs.

7.3 The Hybrid Algorithm

Our goal in creating a hybrid algorithm is to "merge" the best qualities of both adaptive merging and database cracking. In particular, we strive to maintain the lightweight footprint of cracking, which imposes a minimal overhead to queries, and to combine it with adaptive merging's approach of creating and then merging runs, which exploits large memory and ample CPU for fast convergence and enables continuous control of memory and CPU investments.

Data Structures

Before presenting this hybrid algorithm, we first describe the underlying data structures used in the implementation. Each logical column in our model is represented by a pair of parallel arrays that contain row identifiers and key values. Two types of data structures physically organize these pairs of arrays: one partitions them by row identifier then cracks each partition by value and one represents merged ranges of key values. The former represents initial "unsorted" data, and the latter, a partial index over queried ranges of values. These are reminiscent of adaptive merging's runs and final merge partition except that both the ID and value partitions are initially not sorted on keys. As queries are processed, both types of partitions are "cracked" upon the query's keys. Each ID partition uses a table of contents (shown as a small triangle in Figure 7.1)

to keep track of the boundary keys it contains. Finally, a single master table of contents — the adaptive index itself — keeps track of the both id and value partitions (shown as a large triangle in Figure 7.1). It is this adaptive index that is updated as a result of processing queries from a workload.

7.3.1 Algorithm

Assume an input column C and a sequence of range selections over C. We distinguish between the first query and the rest of the queries in the sequence. The first query creates and populates initial data structures. Every query thereafter only performs merging and cracking actions, increasingly refining the adaptive index.

The First Query

The upper third of Figure 7.1 sketches, from left to right, the processing of a first query requesting values between a low bound L_1 and a high bound H_1 from C. This is as follows.

- Column values extracted from the main table are the input to the algorithm; at this point, row identifiers are implicit rather than explicitly represented in data structures.
- Column values are physically partitioned by copying values of C into ID partitions of size p, and an explicit row identifier is associated with each value. At this point, these row ID/value pairs are partitioned only by row ID.
- Each ID partition is physically reorganized via cracking, i.e., (sub-partitioned by value). Tuples with values lower than the queried range are moved to the beginning of the ID partition; tuples that fall into the range of interest are moved in the middle; while those with bigger values are moved to the end of the partition.
- Qualifying values are sequentially moved (along with the respective IDs) from each ID partition to a new value partition created for this query. The relevant ID partitions' table of contents, and also the adaptive index are both updated if necessary.

In this way, each initial ID partition is cracked in place, building a table of contents over each partition to guide future accesses. In Figure 7.1, the dotted

lines in each partition reflect the information stored in the respective index regarding value positions. The adaptive index maintains information about the relevant value ranges, e.g., for each value range whether these values are already merged into the adaptive index and, if so, which value partition holds these values in which positions.

Rest of the Query Sequence

Every query thereafter is required only to leave the adaptive index with all qualifying values merged and in a "logically contiguous" area.

For example, the middle third of Figure 7.1 shows, from left to right, the processing of a second query requesting values between a low value bound L_2 and a high value bound H_2 from C: The algorithm starts with two searches on the adaptive index to locate the areas where low and high bounds fall. We can distinguish between a number of cases.

One scenario is that both bounds fall within a key range that has already been fully merged into a value partition. This happens, for example, when a previous query has already requested an overlapping value range, as shown in Query 2 of Figure 7.1. This query needs all values between L_2 and H_2 (marked with bold lines in the value range area). In this case, all requested values are already in the adaptive index and thus no merging actions are required. The algorithm needs to ensure that all qualifying values are in a contiguous area. To do so, it cracks the value partition that holds these values, reorganizing it in place such that values between L_2 and H_2 are stored continuously. The hybrid algorithm uses the adaptive index to determine where in the chunk it should crack and then registers any cracking actions back to the index such that future accesses to this chunk can exploit the refined knowledge.

If both bounds fall in a single area not yet merged, the hybrid algorithm brings all necessary values from the initial partitions. Such an example is shown in Query 3 of Figure 7.1. It searches the table of contents of each ID partition P_i to find which pieces of P_i should be cracked for L_3 and H_3. After this is done, all values between L_3 and H_3 are in a contiguous area in P_i. These values are then copied into a new value partition. After all qualifying ID partitions have been cracked and all qualifying values copied to the value partition, the value partition is registered in the adaptive index. Notice that each time the algorithm searches an ID partition for a new range of values, the ID partition is sub-partitioned by value (e.g., see the extra dotted lines in Figure 7.1), which further speeds up future searches.

Finally, there is the more general case that the two query bounds are con-

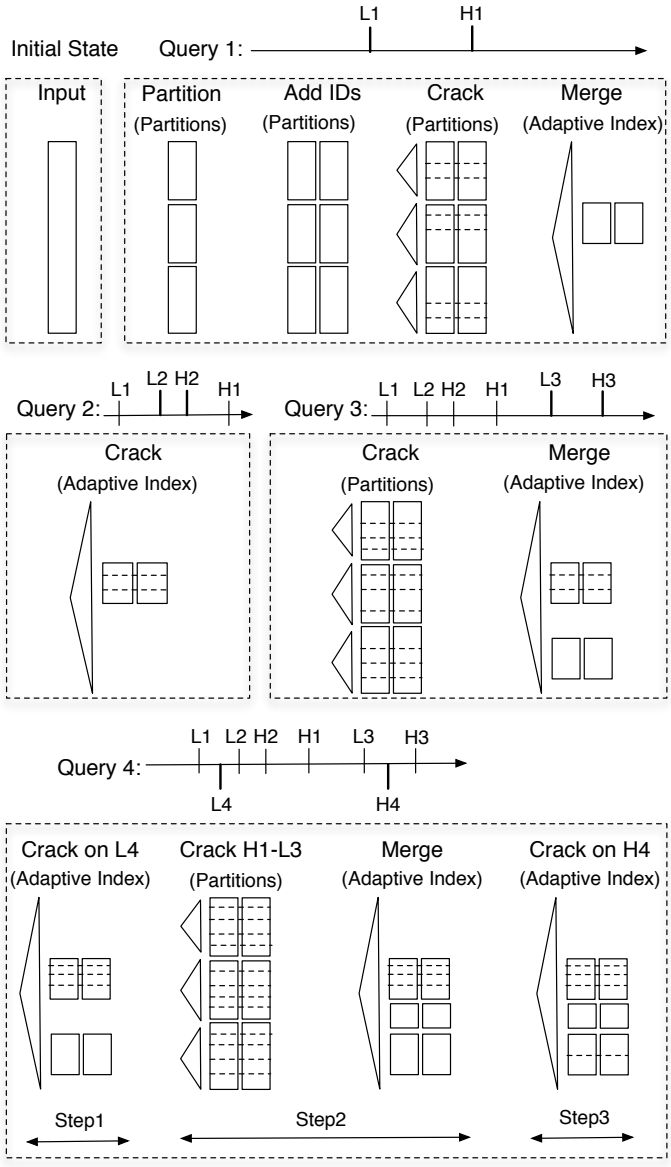


Figure 7.1: Hybrid Example Steps

tained in different value partitions. E.g., the low bound of Query 4 of Figure 7.1 falls in sub-partition a_i of one value partition and the high bound in sub-partition a_n of another. If C possessed the ordering properties of a cracker column, this would mean that all areas in between, i.e., that a_{i+1} to a_{n-1} qualify for the result and we only need to inspect a_1 and a_n. However, in the case of the hybrid algorithm we also need to make sure that all areas in between are actually merged; if not, we must merge the needed key ranges into the value partitions in the adaptive index.

Let us go through the process step by step. The algorithm handles the qualifying areas one by one, starting with a_i. If a_i is a non-merged area, we need to merge all values between the low bound of the query (L_4) and the low bound of a_{i+1} from the initial ID partitions. This is done by cracking the ID partitions and copying qualifying values into a new chunk as discussed above. Otherwise, a_i is already merged, and we can inspect the adaptive index to determine whether values larger than L_4 already form a contiguous area at the end of a_i. If not, we crack a_i accordingly. This is the case for Query 4 of Figure 7.1.

Then, the algorithm checks areas a_{i+1} to a_{n-1}. If an area is not merged, it again uses cracking to merge all qualifying values from the initial partitions into a new value partition and add it to the adaptive index. In our simple example (Query 4), just a single middle area needs to be merged.

Finally, when a_n is reached it is treated as a_i, i.e., if a_n is not merged, then we merge all values between the high bound of a_{n-1} and the high bound of the query (H_4). Otherwise, if a_n is merged, then we crack it based on H_4 to move all values smaller than H_4 to the beginning of a_n.

Hybrid First Query Cracking

In the above description, the first query performs a number of tasks as a side-effect. First, it slices the input column and copies its values into ID partitions. Then, for each ID partition, it materializes tuple identifiers, cracks on query boundaries, and copies qualifying values into a merge value partition.

Here we show how to avoid this last step to make the first query even more lightweight. The observation is that for each partition, the cracking action already writes the qualifying values once. This is to place them in the proper area of the partition. Then, the algorithm copies these values into the value partition, which leads to a second write action. The idea is to have a new cracking algorithm that immediately writes the qualifying values directly into the proper area of the value partition to avoid the second write. Notice, that

we do not need to write these values into the ID partition as well. The reason is that once a value range has been merged into a value partition, the algorithm will never look for it in any ID partition in the future.

In this way, for each ID partition the hybrid performs in one go the creation of the partition (including ID materialization) and the writing of the qualifying values into the appropriate value partitions. This boils down to a filtering action while reading the next p tuples from the input. It creates a new empty ID partition and places every value lower than the requested range in the beginning of the partition, while each value higher than the requested range is placed in the end. Qualifying values are written immediately in the adaptive index's value partitions. The area corresponding to the qualifying values in the ID partition is left unwritten. However, its existence is required to align the positions of the rest of the pieces of each ID partition, allowing the tables of contents to work correctly and guide future accesses.

7.3.2 Insights

The hybrid algorithm replaces the expensive sorting actions during the first query in adaptive merging with cracking actions. In addition, compared to pure cracking, instead of having to operate over a complete array of values in one go, it operates over one small partition at a time, increasing locality and avoiding the delay of first loading the entire column. This makes the first query much more lightweight, resulting in a more fluid adaptation to workload changes without penalizing the query that happened to be the first.

The "overhead" is that if future queries need to merge more values, they cannot exploit fast binary search actions such as adaptive merging can employ to search its initially-sorted runs. Instead, every trip back to the ID partitions is achieved with cracking actions to retrieve the needed values and leave the ID partitions in a state where future accesses can exploit even more knowledge to improve performance. In short, every time we need to merge more values, merging becomes faster and faster as cracking the partitions becomes faster given the accumulated knowledge from previous cracks. This brings a more dynamic and self-organizing behavior with data structures dynamically adapting to the workload when needed.

Similarly, value partitions are not initially sorted. Unless a given query needs its result sorted, there is no benefit in investing in sorting a result area on-the-fly. However, insofar as value partitions represent a partial-ordering and the adaptive index maintains information on the value partitions and the data they contain, the adaptive index permits fast future accesses to data merged

into value partitions. If a future query overlaps with past ones and we need to search within a merged area, we cannot perform binary search actions and we have to perform cracking instead. Again, this transfers the cost dynamically to those queries that actually need to perform the extra work allowing the system to more gradually and incrementally adapt to the workload.

7.3.3 Updates and Multi-column Indexes

Updates

Update algorithms for cracking have been studied in Chapter 4. Given that the final adaptive index produced by the hybrid algorithm is essentially a crack array, the algorithms of Chapter 4 apply without any change in techniques or behavior. The main idea is that updates are modeled as pending insertions and pending deletions. Conceptually, they can be thought of as yet another partition needing to be merged. When an operator processes a given area of the crack array, it also checks whether there are any pending updates for this area. If there are, it directly applies the updates on-the-fly. The trick is to do this in a lightweight fashion while at the same time maintaining knowledge about how values are organized, i.e., without invalidating index information. The algorithms of Chapter 4 exploit the fact that a crack array is not fully sorted and swap/move/insert values only at the edges of the affected areas in an array. Similarly, deletes leave empty spaces at the edges of cracked areas, which are then filled in with future updates on this or neighboring areas.

Multi-column Indexes

Multi-column indexes have been studied in Chapter 5. Those techniques apply directly here, given that the end result of the hybrid algorithm is essentially an augmented crack column. The main idea in Chapter 5 is that the knowledge gained for one column over a sequence of queries is adaptively passed to other columns when the workload demands it, i.e., when multi-column queries appear. The whole design is geared towards improving access to one column given a filtering action in another one, leading to efficient tuple reconstruction in a column-store setting. The net result is that the multi-column index is essentially built and augmented incrementally and adaptively with new pieces from columns and value ranges as queries arrive.

7.4 Experimental Analysis

In this section, we continue with a detailed experimental evaluation. We use a 2.4 GHz Intel Core2 Quad CPU equipped with one 32 KB L1 cache per core, two 4 MB L2 caches, each shared by 2 cores, and 8 GB RAM. The operating system is Fedora 12.

Implementation Details

We implemented adaptive merging and the hybrid algorithm in MonetDB and fully integrated the implementation within the cracking module of MonetDB. This is the first such analysis of these two prior adaptive approaches relying on a complete implementation. In the same way, as with original cracking, adaptive merging and the hybrid algorithm resulted in new select operators that perform, register and maintain any necessary reorganization actions over the data on-the-fly. The complete code base is part of the latest release of MonetDB available via `http://monetdb.cwi.nl/`.

Experimental set-up

The experiments are based on running a sequence of queries with a specific workload pattern over a given data set. The metrics used are (a) the response time for each individual query, (b) the cumulative time to answer the complete query sequence and (c) how fast performance reaches the optimal level, i.e., retrieval performance similar to a complete index. We purposely keep the queries simple to focus on the adaptive behavior of adaptive indexing and how the hybrid algorithm can improve it. Behavior for more complex queries follows the same patterns as in Chapter 5. Here, we use queries of the following form.

```
select a1 from R where a1>low and a1<high
```

We always compare our adaptive indexing methods against two classic approaches. First, we compare against the simple scan of a table without indexes. This represents performance in the absence of investment in index optimization, the typical alternative of a system that has not been prepared for a given workload. Second, we compare against a table with an appropriate sorted index created prior to query processing. In our implementation, the first query performs a complete sort of the data such that all subsequent queries may employ binary search. This case assumes sufficient knowledge and idle time prior to query processing that appropriate indexes can be prepared.

To meet the motivation for adaptive indexing, we will test all methods using a completely dynamic scenario, i.e., a scenario where we do not have the luxury of a-priori knowledge or idle time to prepare indexes ahead of time.

7.4.1 Random Workloads

In our first set of experiments, we use a column of 4×10^8 tuples with randomly distributed integer values in $[0, 10^8)$. We fire 10^4 queries, where each query asks for a randomly located range with 10% selectivity. In general, such random scenarios can be considered the "worst cases" for adaptive indexing as a widely spread access pattern requires many queries to build-up dense index information. We will see in the next section how more focused workloads result in a much faster optimization of the relevant key ranges.

In principle, all strategies (other than scans) eventually result in a fully sorted storage. They differ in *how* they implement the required $n \log n$ comparisons and *when* they schedule them across the workload. The major trade-off is low overhead over a simple scan for the first queries vs. fast convergence to a complete index.

Figure 7.2 depicts the cumulative average response time $\bar{T}(i) = \sum_{j=1}^{i} t(q_j)/i$ for $i \in \{1, \ldots, 10^4\}$. Part c) shows the whole range of 10^4 queries, while parts b) and a) magnify the results for the first 200 and first 10 queries, respectively.

Scan and Sort

Figure 7.2a highlights that sorting makes the first query almost ten times as expensive as a simple scan. Figure 7.2b reveals that, in terms of cumulative costs, sorting in this case only pays off once more than 24 queries have to be answered. Figure 7.2c shows how with long query sequences the cumulative average response time with scan stabilizes at a high level, while with sort it decreases linearly.

Scan has a relative stable performance for each individual query as it always does the same job. Only the first query is more expensive as it needs to also load the data from disk. On the contrary, the complete sort method has a high start-up cost to fully sort the column, but as of the second query it enjoys the optimal performance, which is multiple orders of magnitude better than that of the simple scan (the 100% curves in Figure 7.4 demonstrate the per query costs).

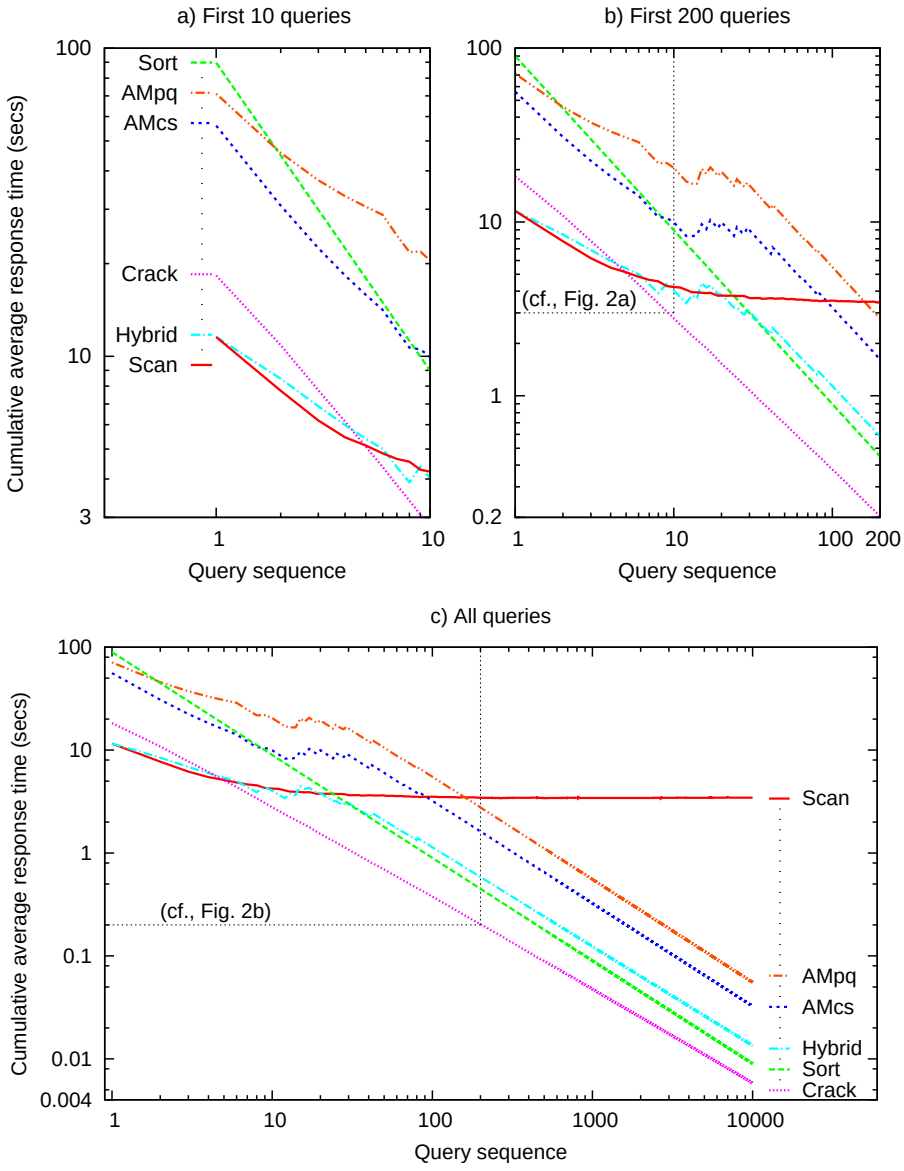


Figure 7.2: Improving Adaptation with the Hybrid

Adaptive Merging

We prepared two implementations of adaptive merging. They differ in how they merge tuples from the initial partitions to the final column. The first one, *AMpq*, uses a traditional priority queue mechanism that has proved to work very well in the context of disk-based paged row-stores. However, in our column-store engine using arrays for in-memory processing, this implementation appeared to be sub-optimal. In short, a large number of runs is required to limit the runs to CPU cache size for efficient sorting. However, merging this large number of runs requires a large priority queue that exceeds the CPU cache size. Perhaps more importantly, a large fan-in interacts poorly with virtual memory and address translation. We plan on a deeper analysis in the future, including multi-level merging.

Consequently, performance suffered from too many cache misses due to the inherently random accesses, both inside the priority queue itself and when accessing the original runs. To overcome this problem, and using the lessons learned from the MonetDB code base, we exploited a "brute-force" mechanism of copying the qualifying key ranges from all runs successively to the final merge column, and then sorting the result there in place, *AMcs*. Featuring purely sequential access during the copy and random access beyond CPU cache size only in the first steps of the final quick-sort, this turned out to be a significantly more efficient implementation for an array-based column-store.

Figure 7.2a shows that both AMpq and AMcs significantly reduce the costs for the first query compared to a full sort. This is due to sorting all small slices individually such that random data access only occurs inside the CPU caches, significantly reducing the number of cache misses compared to completely sorting the whole column. We use runs of size equal to the L1 cache, which proved to provide the best performance. Absolute per-query performance improves very fast for both implementations (much faster than cracking), helping to quickly reduce the average response times (more evident in Figure 7.4). During the initial part of the query sequence, when a completely new value range is requested we need to perform more merging which is why we see a few peaks in performance. Essentially, the cost of the full sort is amortized over multiple queries in a self-organizing way. (This is even more evident if one looks at the per query response time graphs at the end of this section in Figure 7.4).

Purely considering total sequence costs, with AMpq, the high merging costs eliminate the benefit over sort already after the second query. When running at most 8 queries, AMcs offers lower total costs than sort.

Cracking

Adaptive merging achieves its incremental behavior by improving over sorting for the first query while sacrificing a few queries after that to compensate for a non-complete index. Cracking on the other hand concentrates on minimizing the initialization overhead as much as possible at the expense of a longer compensation period targeting more dynamic workloads and bigger savings in case of focused workloads that do not touch the complete data set. For example, in Figure 7.2a we see that cracking needs only 18 seconds for the first query while sorting needs 90 and AMcs 56. After the first query, though, cracking needs significantly more queries than adaptive merging to refine its storage and index before it reaches (close to) minimal response times (cf., Figure 7.4). However, thanks to its low overhead on the first query, the cumulative average times stay below that of adaptive merging and sorting (cf., Figure 7.2c). When purely considering total sequence costs, already as of 5 queries, the initial investment pays off even compared to a simple scan (cf., Figure 7.2a).

The question that arises then is; How can the ideas of adaptive merging and database cracking be combined to bring the costs for the first queries down to no more than the simple scan, while still maintaining the adaptive behavior that quickly converges to an optimal status?

Hybrid

By sorting only small partitions, adaptive merging reduces the cost of the first query compared to sort. However, the cost breakdown shown in Figure 7.3 reveals that sorting the small partitions is still the largest cost component for the first query with AMcs. Hence, to reduce the overhead of the first query, we need to avoid sorting completely. Since also cracking the whole column with the first query comes at a significant cost, we combine the partitioned approach of adaptive merging with cracking to create our new hybrid approach.

Figure 7.2a shows that cracking only CPU cache-sized partitions comes at *practically no overhead* compared to a scan in the first query. Also for the next few queries, the hybrid algorithm shows no overhead compared to simple scan (see also the per query costs in Figure 7.4). By continuously reorganizing the ID partitions as well as the value partitions according to the requested key ranges, it successively refines its storage and index structures. With this, per query costs in Figure 7.4 successively improve at about the same rate and low merging costs as with cracking. Thus, just like cracking it converges slightly slower than adaptive merging, but with the advantage of generally lower and

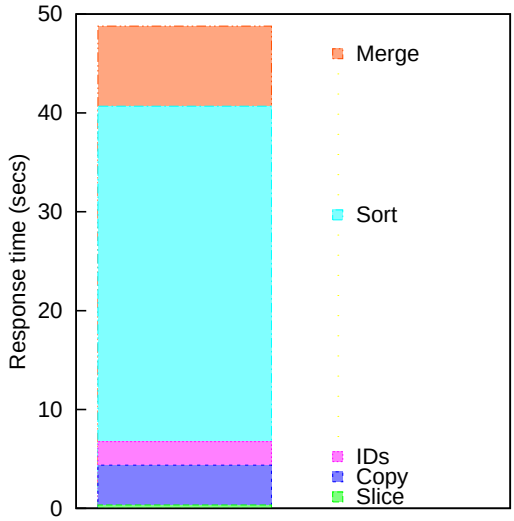


Figure 7.3: AMcs, 1*st* query cost

more predictable merging costs. The major benefit of the hybrid algorithm both over adaptive merging and cracking is that it achieves this adaptive behavior at no extra cost over simple scan in the first queries.

Figures 7.2b and 7.2c show that in the long run, the hybrid sacrifices some performance in terms of cumulative and average response times compared to cracking but at the same time it is significantly faster than those of adaptive merging and of course scan. The major reason is the administrative costs to handle cracking on the large number of partitions required to have CPU cache-sized partitions. We are confident that this cost can be significantly reduced and eventually eliminated by a more sophisticated implementation that fuses partitions to reduce their number as they become smaller, with tuples being merged into the index.

7.4.2 Focused Workloads

Up to now, we studied random workloads. Here, we show how adaptive indexing performs in more focused workloads.

Figure 7.4 shows per query response times $T(i) = t(q_i)$ with queries that

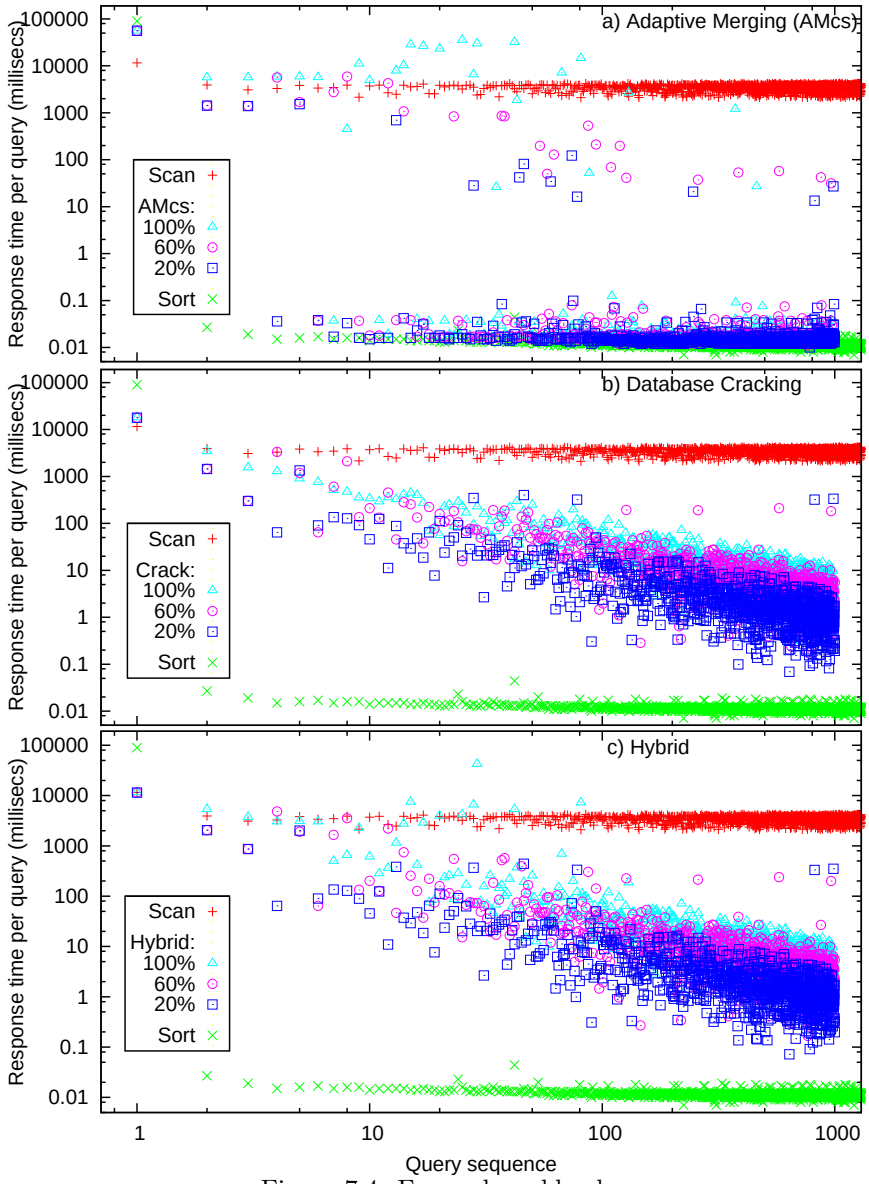


Figure 7.4: Focused workloads

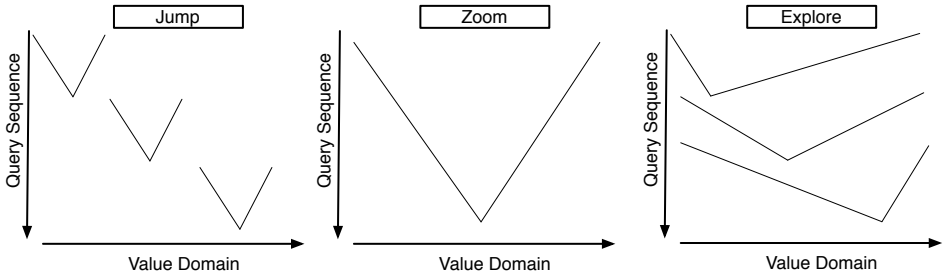


Figure 7.5: Examples of Various Workload Patterns

focus on 100%, 60% and 20% of the data set. The first case is exactly the case we studied so far (same run) representing random queries over the entire data set, while the other two focus on smaller parts. Serving as the two extreme reference points, the scan and sort results are identical across all three graphs. Scan and sort are not affected by the workload pattern, whereas all adaptive indexing methods manage to more quickly adapt as the workload becomes more focused. What happens is that even without a completed index all adaptive methods have already gained enough knowledge for the workload's hot set and can provide the optimal performance even though the rest of the index is essentially empty.

In general, all adaptive indexing strategies show excellent performance and behavior with a variety of focused workloads while the hybrid algorithm maintains its major benefit of essentially building an index for free.

We continue by studying another set of detailed experiments, focusing on individual query performance. In particular, we study various kinds of non-random workloads, i.e., cases that express a particular non-random query pattern. We show that in such cases adaptive indexing techniques achieve even better performance by being able to more quickly reach a fully optimized status, accentuating benefits over traditional techniques.

Up to now, we studied simple focused workloads where the pattern was that we simply restricted our interest to a specific part of the domain, i.e., to $x\%$ of the possible ranges we could query. Here, we study more complex and general patterns where the focus gradually shifts into target areas based on a specific pattern. These kind of scenarios represent more realistic workloads where the user explores and analyzes the data based on on-the-fly observations.

Jump Patterns

In this experiment, we analyze a workload with a jumping focus, i.e., the focus is initially on 20% of the data set, then after 1000 queries it jumps to a different 20%, then to a different 20% and so on. This represent a behavior where the user will eventually study the whole data set but we do not need the complete data set optimized in one go or at all times. The left part of Figure 7.5 shows such a workload; as the query sequence evolves the workload focuses on a small area of the domain and then the focus jumps to a different area.

Zoom Patterns

The zoom workload pattern reflects a zooming behavior where progressive understanding during query processing leads to the actual point of interest, i.e., the workload stepwise zooms into shrinking areas of interest. The middle part of Figure 7.5 shows such an example. Here, the first 2000 queries are randomly spread over the whole key range, the next 2000 queries focus on only the center 80% of the key range, the next 2000 queries focus on the center 60% and so on, until in the $5th$ step the target area has shrunk to the center 20% of the key range.

Exploration Patterns

Finally, we mimic an exploratory workload by combining the Jump and Zoom ideas as follows. The first 500 queries explore the whole key range. Then, in steps of 500 queries, the target is successively limited to the first 50%, 40%, 30% and finally 20% of the key range. Thereafter, 500 queries again consider the whole key range, before a next sequence of 500-query steps zooms into 50%, 40%, 30%, 20% at a different location in the key range. We repeat this 2 more times, i.e., 4 times in total. This workloads represents a quite realistic exploration of the data set where the user continuously zooms in and out of interesting areas trying to interpret the data. The right part of Figure 7.5 reflects such a behavior. Again, at any given time, we only need part of the data set optimized.

Discussion

Figures 7.6, 7.7 and 7.8 show the results. They are meant to provide a high level visualization to provide insights of how adaptive indexing works in a variety of

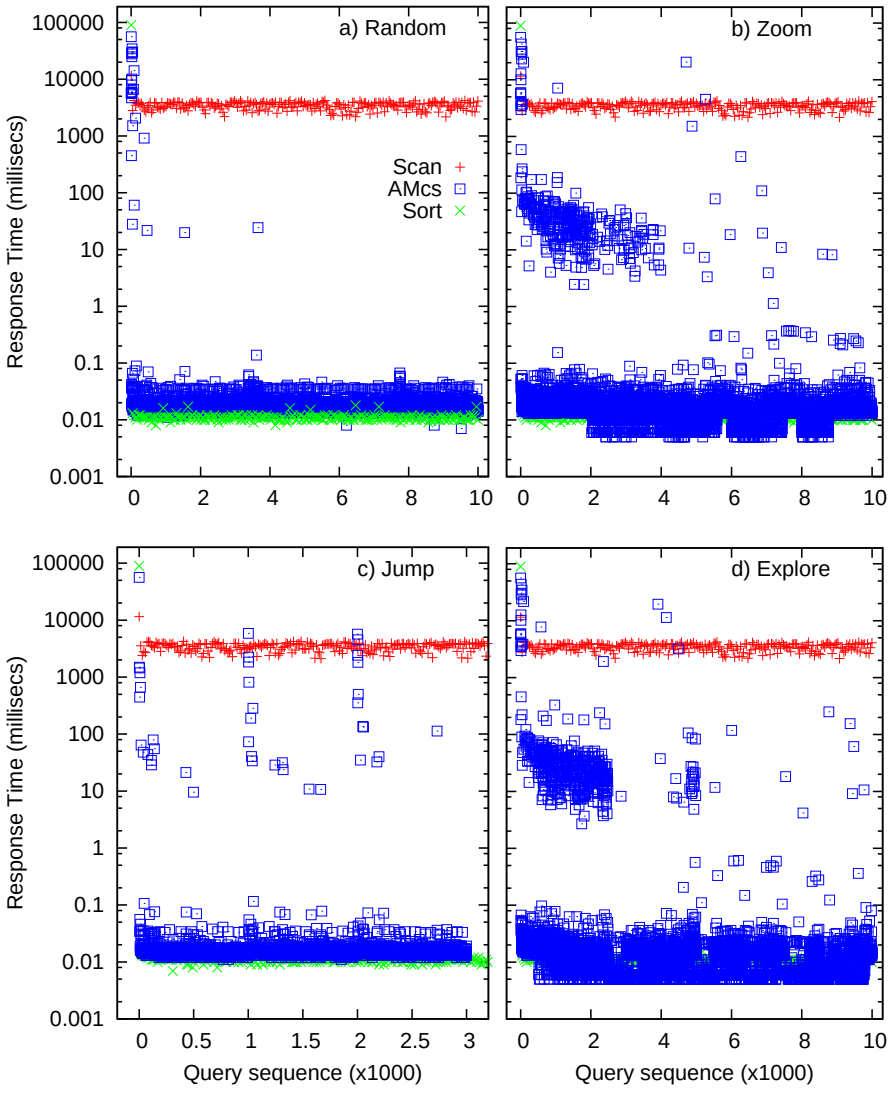


Figure 7.6: Adaptive Merging for Various Workload Patterns

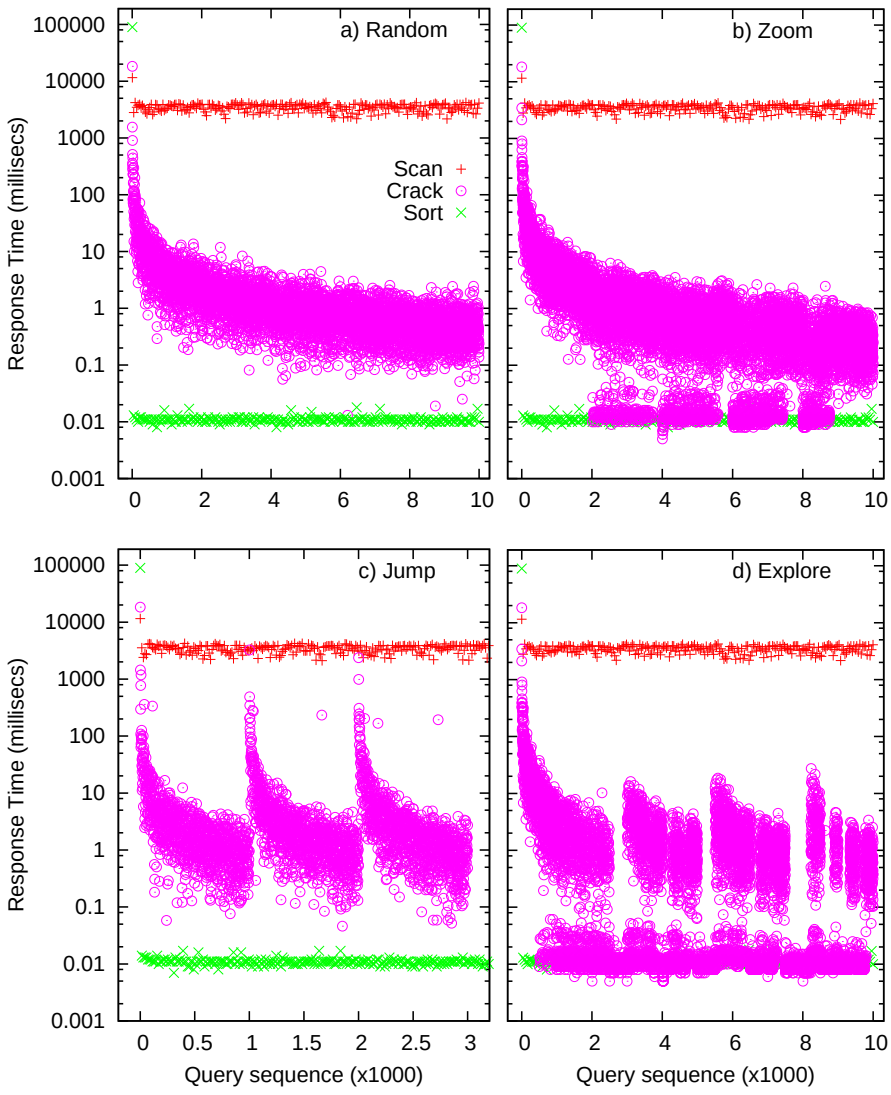


Figure 7.7: Selection Cracking for Various Workload Patterns

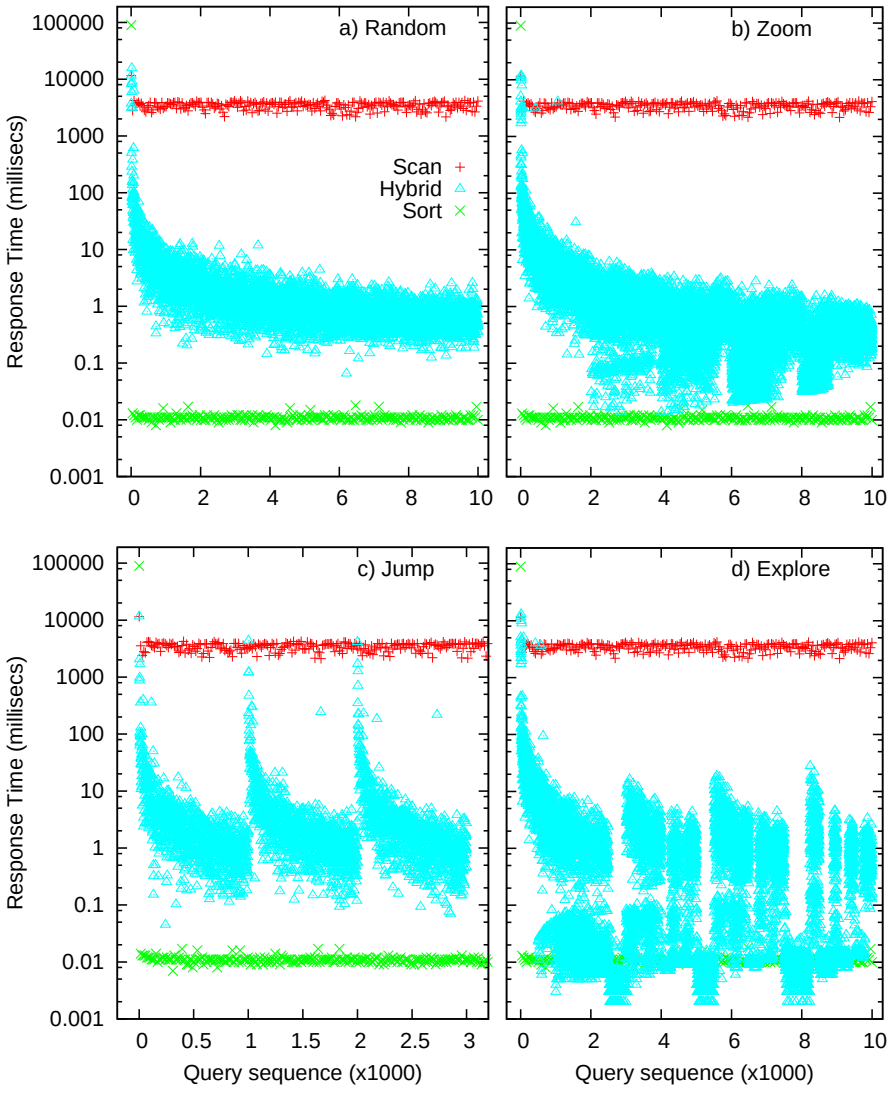


Figure 7.8: Hybrid for Various Workload Patterns

scenarios. For each workload pattern we report all adaptive indexing techniques in comparison to the baselines of a priori sorting and on the fly scan.

Sort and scan are *insensitive* to the workload and thus they provide the same performance across the range of workloads tested. For example, scan will always scan the same amount of data while sort will always do a complete sort to exploit binary search from there on. Thus, scan has always the same high cost, while sort will always pay a heavy initial cost. Adaptive indexing though provides a very different behavior trying to always exploit the workload patterns in order to improve. This is true for all our adaptive indexing techniques with the hybrid having the edge due to the more lightweight adaptation as we discussed in the main paper. We do not provide figures that focus on the initial part of the sequence, comparing initial costs, as this is the same behavior seen before. The goal here is to present an overview of how the adaptive indexing algorithms adapt to the changing workload patterns. For example, notice how for the Zoom workload all adaptive indexing techniques improve faster (in terms of queries processed) to the optimal performance levels compared to a random workload as well as improving even further. The fastest query in the Zoom workload is one order of magnitude faster than the fastest in the Random one. By focusing the workload on specific key ranges, adaptive indexing can improve its knowledge over these key ranges more quickly. Similar observations stand for the Jump and Explore workload patterns as well. In these cases, the shift from one key range to another is visible by a few high peaks each time representing a workload change. Once the workload focuses again, performance improves very fast and in most cases performance stays way faster than the plain scan approach. Without any external knowledge and completely in an automatic and self-tune way performance improves purely based on the workload.

7.5 Summary

Adaptive merging extends the ideas of cracking in a very interesting way targeting mainly disk processing trying to minimize the adaptation period. The new techniques presented in this chapter make the first step in expanding the scope of adaptive indexing even further by trying to blend ideas and concepts from both adaptive merging and cracking. They offer an additional choice that combines advantages and avoids disadvantages: *hardly any burden is added to a scan!* This suggests an interesting new approach to physical database design. The initial database contains no indexes (or only indexes required in support of uniqueness integrity constraints). Query processing initially relies on large

scans, yet all scans contribute to index optimization in the key ranges of actual interest. Due to the fast index optimization demonstrated in our implementation and our experiments, query processing quickly transitions from relying on scans to exploiting indexes. Due to low burden required to create and optimize indexes, query processing using this approach to physical database design should offer the best of both traditional scan-based query processing and traditional index-based query processing, without the need for explicit tuning or for a workload for index creation.

Adaptive indexing opens a whole new research area. The work in this thesis has only scratched the surface of it, demonstrating the feasibility and the potential. Several issues are open for detailed studies. For example, the work in this chapter concentrates on the index initialization overhead, assuming mainly a memory resident DB or at least one where the hot set mostly fits in memory. There are several other parameters to consider when comparing all these adaptive indexing techniques, i.e., concurrency control, updates, recovery, scalability for large data, etc. In many of these cases, a more active method than pure cracking can potentially be more beneficial. For example, pure adaptive merging with its very active sorting steps in the beginning of a query sequence, significantly reduces the adaptation period resulting in a highly optimized index very soon in terms of numbers of queries needed. This helps with concurrent queries since the index needs no further reorganization and thus there are no conflicts. It also requires less effort in continuously rewriting a disk based index every time we reorganize it as there are less reorganization periods. In addition, low-level implementation details could potentially affect the performance picture, i.e., adaptive merging could gain a lot from a multi-level merging approach.

It is evident thus, that there is a plethora of paths to investigate towards the ultimate adaptive indexing method that can cope with any kind of workload and system set-up. The first hybrid approach presented in this chapter has all the hooks to become the vehicle towards this ultimate goal and properly balance the relevant tradeoffs, e.g., initialization investment, adaptation speed, scalability, etc. There is a more detailed discussion on these issues in the next chapter. In simple words, with the hybrid we can dynamically schedule how many reorganization actions we do. The trigger does not have to be a single query, but the general status of the system and the workload, resulting in more active or more lazy reorganization periods to fit the system and workload.

Having seen in the previous chapters a detailed discussion on the cracking architecture, the algorithms for updates and join processing, as well as techniques inspired by cracking, merging and hybrids in this chapter, we move on to discuss future research directions in the next chapter.

Chapter 8

The Big Picture

The goal of this thesis is to open the ground towards truly self-organizing database systems, i.e., systems that without the need of human administration can cope with and adjust to dynamic environments with continuously changing workloads. The system should just be given the queries and data in a declarative way and should internally make all performance related decisions, automatically and dynamically, leading to simple to tune and thus scalable systems to meet the modern challenges, e.g., scientific databases, web-based applications, etc.

We introduced database cracking, a new architectural paradigm in the context of column-stores where the system internally and continuously makes on-the-fly decisions to physically reorganize the data so that access patterns match the workload needs. Cracking does not require any expensive preparation steps or a priori workload knowledge. It purely and automatically reacts with lightweight actions on the data and query workload.

What we Did

In this thesis, we introduced the basic cracking concepts. We designed and implemented a cracking architecture over an existing system successfully integrating cracking in the complete software stack of MonetDB. We demonstrated a clear self-organizing behavior even during completely random workloads. In addition, we provided an in depth study and solutions for two of the most crucial issues, i.e., updates and tuple reconstruction. By exploiting novel self-organizing algorithms, cracking maintains its properties even under frequent, random and heavy updates, while via partial sideways cracking we show how a

cracking DBMS can achieve late tuple reconstruction similar to that of a presorted column-store but without the restrictions of requiring a priori workload knowledge, idle time to prepare and infrequent updates. Finally, we demonstrated how cracking knowledge gathered during query processing, can be exploited and even enhanced by crack joins, that on-the-fly further reorganize join inputs in a cache conscious way.

A New Challenging Research Area

This thesis represents a first crucial step towards a completely autonomous DBMS kernel but most importantly it opens a challenging research path which essentially calls for reconsidering many design aspects of modern DBMSs. Cracking fundamentally changes the way data is stored and accessed, and thus many traditional core database techniques and ideas are not sufficient and need to be thought again towards a dynamic system. In the rest of this chapter, we discuss in detail a number of this research challenges, e.g., optimization, concurrency control, external cracking, compression, cracked histograms, distributed and multi-core processing, etc. Where appropriate, we provide sketches of promising initial directions/solutions and how these would fit into, affect or benefit from the current design.

8.1 Cracking Operators

In this thesis, we studied new cracking operators for selections, tuple reconstructions and joins. Modern databases contain numerous operators to achieve their advanced functionality. Cracking is completely compatible with non cracking operators, i.e., in the TPC-H experiments of Chapter 5 the cracking query plans used the normal DBMS operators for aggregations, group by, order by, etc. However, it is clear that if we want to exploit the full potential of this new research path, we need to dig deep into the kernel. Thus, the open question is; which other operators can we redesign and improve via cracking?

The core principles of a cracking operator is that it physically reorganizes data such that the input columns will provide better access patterns in future crack operators. This happens while processing queries and it is based on what the queries need which gives the property of self-organization. For example, assume the group by or the order by operator. Both have an inherit clustering nature that could be exploited in a cracking way, i.e., bring all tuples that belong to the same group physically closer across multiple columns.

The challenge is not only in designing new individual operators, but most importantly it is in providing a balanced direction to avoid side-effects of one operator on another, e.g., physical reorganization performed in a certain way in an operator $o1$ may hamper performance in another operator $o2$ due to the different way of reading and analyzing similar data. Investigating the various tradeoffs is of core importance as well as designing alternative solutions where workload based partial crack replicas of individual data pieces can provide the ability to introduce multiple (combinations of) patterns over the same data for different usages in the hot set. New operators can also be designed to exploit piece-wise processing along with on-the-fly physical reorganization in the spirit of partial sideways cracking.

8.2 Hardware and Data Sensitive Cracking

In the current cracking architecture physical reorganization happens for each incoming query q based only on what q needs. This allows the system to adapt and continuously match the workload. Cracking reacts to every query so that it can provide immediate adaptation as opposed to static approaches that wait long intervals by monitoring the performance before applying an action. However, it is clear that several interesting parameters can be considered to further improve both the performance and the adaptive behavior of cracking. A very interesting research path is to study hybrid approaches, allowing to cleverly understand situations where it is of benefit to invest a bit more, a bit less or even to completely avoid any actions. For example, how can we change/optimize cracking for situations where:

- candidate data for cracking does not fit in system memory or CPU caches?
- candidate data for cracking comfortably fits in CPU caches?
- administration cost of crack pieces is becoming visible?
- we can partially predict future query patterns?
- there is some idle time?

This is just a glimpse of what one may consider and of course again the challenge is not only to find and analyze individual solutions in isolation but to integrate them in the complete architecture. In the remainder of this section, we discuss some of these very interesting topics.

8.2.1 Cache-conscious and Opportunistic Cracking

A side-effect of the continuous physical reorganization is that pieces become smaller and smaller that will eventually fit in the CPU caches. We have already seen how this can affect cracking query processing as part of our join study in Chapter 6. However, there are even more situations to study.

For example, a piece that is small enough to fit in the CPU cashes can be immediately sorted in one very fast action. In some sense this can be seen as *opportunistic cracking*. There are several good reasons to do this, but the implications need to be studied carefully.

The benefit is that, at a minimal cost, we are further preparing the physical design so that future queries will have less work to do. We are piggy packing "more cracking" with a single read operation of the relevant data. In other words, since we are reading the data anyway, let us do something more with it than just a binary plain cracking action. Any action within the cache is quite cheap anyway. However, this requires very careful study as there are also numerous side-effects to consider, i.e., updates and alignment.

Updates

Updates is a strong point of cracking as we show in Chapter 4. By exploiting the fact that values in a cracker column are only clustered as opposed to fully sorted, we can efficiently update the columns with a practically non visible cost, maintaining the self-organizing behavior. Sorted pieces over a column will force updates to move around big chunks of values in order to maintain nicely dense and packed columns, significantly increasing the cost of updates. However, there is a very interesting balance and tradeoff to study here given how often an area of a cracker column is updated and how often it is used for reads. Furthermore, in the spirit of the continuously adaptive behavior of cracking we can simply mark a previously sorted area in a cracker column as non sorted once we realize that this area needs to be updated and the expected overhead is not justified. More cracking knowledge over this area will be rebuild as future queries access it again. This is similar to the forget strategy of Chapter 4 but now it seems as a more attractive direction given that it helps to avoid a potentially big cost.

Alignment

Alignment is also a crucial topic here. If we sort a small piece in one column $A1$, then this means that if the workload demands the respective area from another column $A2$ of the same table, then we would have to sort this area in

$A2$ as well. This is necessary to guarantee alignment and optimal access patters during tuple reconstruction as we studied in Chapter 5. This is not necessarily a bad thing; it all depends on the current workload patterns. Selectively sorting small pieces also reduces the number of logical pieces in a cracked column, which in turn reduces the administration cost. Following the self-organizing nature of the cracked DBMS vision, sorted pieces can become unsorted in the future, e.g., because this specific area is frequently updated.

8.2.2 External Cracking

Similarly, data pieces that are large enough so that they do not fit even in main memory can result in poor performance when reorganization happens with the conventional cracking algorithms. The main issues to consider, analyze and optimize are the read/write patterns of cracking. In principal, cracking has sequential read/write patterns, i.e., it reads sequentially from both edges of the targeted area and carefully swaps values if necessary. However, there is plenty of room to improve for external processing by (a) minimizing the I/O overhead and (b) minimizing the write cost.

Optimized cracking for out of memory processing is needed to properly balance the access and reorganization costs in respect to the anticipated benefits. Understanding and reacting to these concepts should be an integral part of every cracking element in the database kernel. There are several options to consider.

External Algorithms

One option is to pursue specialized external cracking algorithms, i.e., once a piece that is about to be cracked is identified to be bigger than a threshold related to the memory size, then different algorithms than the current ones will run. Taking care of ensuring optimal access and write patterns is the key.

Every cracking action results in changes in the underlying data. These changes need to be written back. As the data grows, more changes are to be expected leading to more write effort and to a substantial cost.

Multi-pass Cracking

One way, to significantly reduce this write overhead is to do less cracking or to do more cracking but write back only once. For example, a *multi-pass* cracking approach might be more beneficial than the current one-pass algorithms that strive for minimal read actions. We can in one go create multiple pieces. This

has proven very efficient in previous MonetDB research, i.e., (Boncz et al., 1999; Manegold et al., 2002) and is expected to be a promising alternative here as well. The net result, is that we immediately end up with a cracker column that is already cracked in multiple pieces of small enough size, significantly improving future queries. This multi-crack action can be piggy packed to the first query ensuring that its overhead is maintained at a desirable minimum to avoid penalizing single queries.

Forgetting

Another interesting direction is that of forgetting what cracking actions we just did to avoid the need to write the changes. For example, once a crack column is sufficiently cracked it might be of benefit to operate as follows.

- (a) Load a needed area to memory.
- (b) Let a series of queries crack it to get the maximum performance.
- (c) In the meantime write any changes only to memory and not to disk.
- (d) When the time comes to write to disk simply *ignore* all changes and skip the writing part.

This essentially means that future queries need to crack the same area of the column again. However, this is only a local action that helps avoid the bigger cost of writing. Of course the proper balance and tradeoffs need careful analysis. In addition, note that multiple positive side-effects can play a significant role. Forgetting helps to not end-up with a cracker column that is cracked into numerous small and "meaningless" pieces which lead to an increased administration cost and to more effort during updates (see also discussion in Section 8.2.5).

Flash-based Cracking

Hardware trends create even more interesting paths for external cracking. For the growing market of flash drives, reads is not a bottleneck while writes can be up to a point taken care of by using batch writes. This way, specialized flash based cracking seems as a promising approach to tackle external cracking by collecting multiple cracking actions and write them back in a single action.

Divide and Conquer

Divide and conquer approaches have always been the cornerstone of external processing. Recently, researchers from the HP Labs in Palo Alto, Goetz Graefe and Harumi Kuno, where inspired by cracking and invented adaptive merging which from a high level point of view can be described as an incremental external sort in the spirit of database cracking.

As part of an ongoing joint work with Goetz Graefe and Harumi Kuno, we have designed and implemented adaptive merging over a column-store and we have made the first steps towards combining these ideas with the cracking philosophy as discussed in Chapter 7. The hybrid adaptive indexing algorithm, introduced and studied in this chapter, reduces the cost of index creation and incremental index optimization to that of scanning a database table without an index. This enables further research into adaptive indexing and its application in real world scenarios. Here we briefly discuss some of these topics.

Alternative Hybrid Designs

The most exciting is perhaps the direction of filling the "gap" between adaptive merging and the hybrid algorithm in a dynamic way. Adaptive merging makes a fair number of investments during the initial queries. This is far less than a complete index creation but still much more than the hybrid or initial cracking algorithms. The reward of early investments is rapid convergence to a fully optimized index. The hybrid algorithm, on the other hand, is optimized to provide the most lightweight initialization cost possible. But what happens when we actually can afford to spend some more time during the first steps of workload adaptation? For example, not as much time as adaptive merging would require but more than what the hybrid algorithm needs. The research question to answer then is: Can we devise an algorithm that adaptively adjusts its investment policy depending on various system and workload conditions? To put this in more technical terms, the hybrid algorithm cracks the initial partitions while adaptive merging sorts them. A future dynamic algorithm could decide on the fly how much to reorganize the initial partitions to exploit as much as possible the available resources, effectively inserting more knowledge into the system and reaching the optimal performance faster. The same stands for the value partitions in the final adaptive index, where the hybrid algorithm now simply copies the values while adaptive merging fully sorts. The above directions will allow for even more hybrid designs that blend even more the good properties of adaptive merging and cracking.

Alternative Implementations

Low level implementation details can change the performance tradeoffs dramatically. We plan to continue our research by investigating design tradeoffs and side-effects in our adaptive indexing techniques. For example, adaptive merging and the hybrid algorithm can benefit from a more advanced table of contents that keeps track of a global view of available values in the various partitions, allowing to completely avoid touching partitions if no qualifying value is to be found there. This might require integration of zone and partition filters (Graefe, 2009) with partitioned B-trees and our hybrid algorithms.

Page-based Vs. Columnar Storage. More importantly, we plan to study in more detail the side-effects of using a page-based storage scheme as opposed to a columnar one. The design and experiments in this paper are based on a column-store system which allows us to have an apples-to-apples comparison of all techniques with a complete implementation inside the original cracking module of MonetDB. However, a number of design issues are expected to be handled more efficiently in a page-based system as opposed to an array-based one. For example, for adaptive merging and the hybrid algorithm, we can probably make the initial part of the query sequence a bit more lightweight by more easily removing data from the initial partitions once they are merged. On the other hand, in our current design, we have exploited the columnar structure to provide better locality and access patterns by adopting several implementation tactics from original cracking. It will be an interesting and useful study to investigate how the various tradeoffs balance also depending on the workload.

8.2.3 Hybrid Cracking

As we said earlier in this section, a more hybrid version of cracking will be very interesting to study. What this means is that for certain queries and given the system status it might be of benefit to invest a bit more or a bit less. The example we had with opportunistic selective sorting in Section 8.2.1 falls in this category as well.

Here, we are considering similar ideas but purely from the workload perspective. Observing how the query patterns evolve over time, i.e., how the cracker indices build up, we can possibly predict/mark certain areas of a cracker column as areas of high interest for a given period of time. In fact, this task alone is a hard challenge. Assuming such knowledge is collected, then when we come to crack this area in a future crack operator, we can possibly invest slightly more effort in physical reorganization to gain much more knowledge. A very simple

example, that comes out of our experiments with numerous alternatives for the basic cracking algorithms, is that if we are bound to crack an area into three new pieces, then it pays off to crack it in three pieces in one go with a dedicated algorithm, than running twice our best algorithm that creates two pieces at a time.

For example, in Chapter 6 we studied active cracking in the context of crack joins. There, we found out that it pays off to invest more effort in cracking the columns into smaller pieces during a join. The benefits during joining smaller and aligned pieces are far more significant than the overhead of cracking, while a careful cache-conscious design ensures that we always remain within an optimal balance of investment versus overhead and gains observed.

In any case, exploiting statistics, collected at run time in the past, is interesting to study; how they could affect cracking decisions possibly by even sacrificing the current query's performance is an open topic. One could go as far as arguing that in certain cases it could be of benefit not to crack at all a given area in a cracker column, i.e, to simplify upcoming updates, or to reduce future alignment costs.

The challenging part is to allow cracking to take more educated decisions but at the same time do not introduce any delays or significantly penalize single queries. Otherwise, this would be completely off the whole purpose of the cracking ideas.

8.2.4 Idle time Cracking

Similar to the above discussion, exploiting possible idle time, is an interesting topic. Cracking is designed for environments without idle time but if some idle time occurs we should be able to exploit it. Assuming the light-weight monitoring and predicting technology necessary for the above scheme as well, then at idle time the system can autonomously satisfy some of the pending hypothetical requests by initiating cracking actions to the relevant columns. As we have seen in many of our experiments, even a single crack action can significantly benefit a subsequent access to the same column so even a minimal idle time investment can have a big impact.

8.2.5 Forgetting

During the whole length of this thesis, we are talking about *learning* via continuously cracking. Here, we argue that *forgetting* might also be of great interest.

By forgetting we simply mean to forget the knowledge we have for a certain area (value range) in a column. This can be beneficial for a number of reasons.

Administration Costs

Administration cost is a crucial issue. The more pieces in a column, the higher the administration cost. Even simply searching for the relevant pieces to crack within the tree of a cracker index, can be of significant cost when pieces grow a lot relatively to the actual data and query costs. We have already seen an interesting aspect of this issue in our crack joins discussion where the administration cost during a crack join was enough to overshadow any benefits. Using super pieces to reduce the administration cost during the crack joins is a form of temporarily forgetting that we actually know about more pieces. What forgetting means will definitely vary from operator to operator also depending on implementation and hardware details.

Updates

Another crucial reason to forget cracking knowledge is to simplify and speed up the adoption of updates. The less pieces in a column, the less effort we need to insert X updates in this column. A very interesting problem is how all this can happen dynamically, i.e., while we update a certain area of a cracker column during a join or a select, we on-the-fly decide that we are going to drop information for this piece or pieces to speed up this update action or part of it.

Reverse Cracking

Alignment and updates can be reasons that lead to the very interesting path of *reverse* cracking. The idea is that we can reverse one or more cracking actions by reversing the actions of the physical reorganization algorithms.

For example, assume we already cracked n times a column A_1 of a given table. Then, more queries come that need to search or even update multiple of the other columns of this table. We have to pay the alignment cost for all remaining columns and updates will run a little bit slower. We have shown that all this happens in an incremental and self-organizing way. However, there are even more exciting paths to study. It could be that aligning k columns with the cracking actions of a single one, A_1, could be slower than the cost of reversing the previous cracks on A_1. Even a hybrid approach is possible, i.e., reverse part of the cracks which leads in needing to run only part of the alignment.

Reverse actions can also be logged in the tapes so that the alignment scheme is consistent.

8.3 Multi-query Processing and Transactions

The challenge here comes mainly from the fact that every cracking operator changes the physical store, e.g., multiple concurrent queries will have conflicts when touching similar data and rolling back transactions should result in a correct physical state.

Conflicting queries can be resolved in multiple interesting ways. For example, properly ordering the queries based on the bounding condition allows the batch of initially conflicting queries to operate more and more in parallel as the sequence evolves. We can also temporarily maintain multiple replicas of selected pieces of a map and allow different queries to use them. Queries can also have concurrent access to pieces but scan instead of crack. Here we can exploit cooperative scans. Scanning a small piece is more efficient than scanning the full columns. Even a full sort of small pieces (within the cache) is possible which also eliminates any locking need. For bigger pieces there is a nice trade-off regarding total costs; crack or scan for part or all of the queries taking into account the global workload behavior and investments for achieving a self-* behavior.

Temporarily locking the cracker columns is also possible and leads to interesting directions. Obviously, the first very naive approach is to provide locking at the map level. This effectively allows query plans to run in parallel and only operators needing to crack the same map have to be potentially delayed, i.e., wait for another operator to finish. However, given that maps are cracked in several logical but also physical pieces we can easily provide locking at a lower level. The main idea to exploit is that two or more operators have a conflict only if they need to crack (thus physically reorganize) the same chunk of a partial map or even better the same piece of a chunk. Given that each operator may crack at most two pieces of a map, this provides a lot of flexibility.

Furthermore, it follows the self-organizing nature of database cracking. Given that cracking becomes faster as more queries touch/crack a given map, as more queries need to crack a given area of a map, the locking time needed for pieces in this area becomes continuously shorter. The main idea is that each small data piece created via cracking will be treated completely *independently* providing maximum flexibility to handle all aspects regarding this piece in isolation, e.g., avoid query conflicts, ease transactions, provide optimal access patterns etc. This notion of independent piece wise processing has to be an integral part of

every operator.

8.4 Compression on Cracked Columns

As we already discussed in Chapter 2, compression represents huge opportunity in column-stores. By exploiting the fact that each column is stored separately, we can significantly increase the compression ratio. The open issue is; can we compress cracked columns?

Cracking Compressed Columns

One interesting research path is to device algorithms for applying cracking actions directly on compressed columns. For example, assume a compressed column A_1. Say A_1 has been compressed using dictionary compression and for simplicity assume one dictionary per column. We can extend the cracking algorithms/operators to work directly on the compressed column by on-the-fly decompressing values to apply the necessary steps, controls, etc. Then, the actual physical reorganization action of swapping two values is nothing more than simply swapping the respective pointers to the dictionary. This can become an even simpler procedure if the compression used ensures an order preserving scheme for the dictionary compression, i.e., the code follows the order of the actual values. Then, we can simply operate over the compressed column without looking at the dictionary.

If the compression scheme is run length encoding, then we need to also recalculate the new deltas based on the values we are about to swap. This is straightforward. What is more challenging here is to cope with cases where the various compressed values are not of a fixed size. This is an issue if for example we want to swap two values v_1 and v_2, where v_1 is say 2 bytes and v_2 is 3. Assuming dense columns, then there is no space to move v_2 in the place of v_1.

Crack for Compression

The above problem is the perfect motivation for introducing compression criteria in the cracking algorithms. For example, in our previous example, we had problems moving v_2 in the place of v_1. A compression aware cracking algorithm, after spotting this situation could decide to crack a bit more or a bit less to accommodate the compression needs as well. It could decide to move more similar values close to v_1 in order to accommodate the new v_2. What is really interesting to observe here is that in general the more we crack a compressed

column, the smaller it becomes (assuming run length encoding). This is because similar values continuously come physically closer, leading into smaller deltas and thus requiring less space which can be exploited also for cases like the one in our previous example.

8.5 Adaptive Denormalization via Cracking

Here, we give an example of how cracking can allow us to reconsider basic database design issues. The way data is stored on disk and in memory dictates the way in which it can be accessed and processed. Normalization has long been the textbook gold standard to guarantee integrity constraints, minimize storage requirements and update costs. This way, database systems store data as a collection of relational tables linked via foreign key relationships. The overhead is the significant cost required to join the separate relational tables during query processing. This happens via the join operator which is the most expensive database operator. Even nowadays that database systems can exploit a large collection of specialized join algorithms, The join cost typically dominates the total cost of query plans. In predictable and stable environments where the query workload can be predicted, materialized views can be exploited, having the frequent joins precomputed.

With the advent of partial sideways cracking, as proposed in this thesis, adaptive denormalization becomes a clear opportunity for fully dynamic and unpredictable environments. In addition, column-stores make this research direction viable; they can naturally handle very wide tables at no extra cost.

In adaptive denormalization, partial sideways cracking can be extended such as each map set is populated not only with columns from a single table but with columns that belong to *multiple* tables. In practice, the join operators used in queries determine these "connections" and each given query is answered using a single partial map set. In the same way, that a crack select operator incrementally brings often used data physically closer, these new foreign key joins will physically denormalize the relevant areas and place them in such positions in the maps that they are aligned with the rest of the columns in this map set. All basic concepts of partial sideways cracking are maintained with the underlying storage patterns changing continuously and automatically to adapt such that the physical design is *always denormalized for the hot workload set.*

Each query is evaluated using a single auxiliary "universal cracker map" that holds partial denormalized data and is dynamically populated and partitioned both vertically and horizontally. Once a foreign key join is needed for a given

value range of two tables, the underlying storage adapts such that the joinable tuples are now physically stored/glued together in auxiliary data structures as if we denormalized the relevant portion of the respective tables. Denormalization happens continuously; while processing queries and partially; only for the required/hot data ranges. When necessary, denormalized data parts (outside the hot set) are automatically thrown away to cope with changing workload behavior and storage restrictions.

With this direction, the maps become wider making their maintenance and alignment even more crucial while the partial sideways techniques have to be thought again.

8.6 Cracking Row-stores

All technology introduced in this thesis is in the context of column-stores. The open question is whether all these or part of these ideas can be transferred in a row-store setting.

From a high level point of view we can say that the cracking ideas as such would be very interesting to study in a row-store. The actual technical parts would have to be reconsidered though. Cracking goes deep into changing the way data is stored and accessed. This way, many core cracking techniques depend on the column-store layout.

Reorganizing row-store pages immediately raises numerous challenges. As we already discussed in Chapter 2, row-store pages contain a number of metadata entries, having a specific structure to maintain. Thus, cracking a row-store means that we would need to maintain and reorganize a number of metadata entries. There are two directions here. One is designing efficient algorithms for continuous reorganization of row-store pages and the other is to additionally reconsider the page format for cracked tables to simplify physical actions.

In addition, another crucial point is that we would need to reorganize complete rows at a time even if queries are interested only for part of a table. On option of course, is to let cracking work only on materialized views that hold only a portion of a table at a time. The simple view then incrementally becomes a more powerful data source that also knows about how data is organized. Second, the alignment cost should be carefully taken into account. Reorganizing some extra columns on-the-fly means that no alignment is needed when future queries use them. Especially, in the materialized view world this throws another interesting parameter to consider when creating and using views.

8.7 Adaptive Indexing in Auto-tuning Tools

Adaptive indexing is not necessarily a replacement of traditional techniques. It rather can be seen as yet another tool for when the environment does not fit traditional indexing techniques, i.e., for when the workload changes often and rapidly.

Such a dynamic investment direction as database cracking could find its place in auto-tuning tools that analyze the expected workload a-priori and then create all necessary indexes. A traditional auto-tuning tool might decide which indexes to create immediately, which indexes to prohibit (e.g., in order to avoid all space or update costs), and which indexes to create incrementally as side effect of query processing. An auto-tuning tool could also suggest materialized views and which indexes to create incrementally using adaptive indexing techniques for these materialized views. Finally, an auto-tuning tool could decide for each incremental index how much of an initial investment to make, based both on the objectives of the current workload and on the importance of this index for the expected workload.

8.8 A Histogram for Free

The cracker index contains information on actual value ranges of a given attribute. This is useful information that can be potentially used in many cases. For example, the cracker index could play the role of a "histogram" and allow us to take decisions that will speed up query processing. With a cracker index it is known for a given range *how many* tuples in a column are in that range. This could be a valuable approximate information, e.g., in the case where the given range is not an exact match with what exists in the cracker index.

Traditionally histograms are maintained separately, which leads to an extra storage and operation cost. In addition, they are not always up to date with the current status in the database. On the contrary, with cracking the histogram-like information comes for free since no extra storage or operations are needed to maintain it. An interesting observation, is that here the histogram is a self-organized data structure as well; it creates and maintains information only for parts of the data that are interesting for the users. This is a powerful property for a structure that can potentially affect many aspects of query processing performance.

8.9 Distributed Cracking

Cracking is a natural way to partition data into "interesting" pieces based on query workload. Therefore cracking can be explored in a distributed or parallel setting by distributing pieces of the database to multiple nodes.

For example, each node of the network may hold one or more piece of each crack column. The cracker index can be known by all nodes so that a query can be navigated to the appropriate node that holds the interesting data. One can explore more sophisticated architectures where the cracker index is also distributed to multiple nodes, to reduce maintenance cost (more expensive in a distributed setting mainly due to network traffic creation and delays). In this way, each node has a partial knowledge of the index and a partial knowledge of the data, thus there is a need for the proper distributed protocols/query plans to correctly and completely resolve a query. However, these are typical requirements/research challenges in any distributed setting. The key here is that the way data is distributed is done in a self-organized way based on query workload which we envision that can lead to a distributed system with less network overhead since interesting data for queries will be already together.

Distributed cracking is a wide open research area. It can be explored in the context of distributed or parallel databases. Furthermore, it can be explored in the context of P2P data management architectures where typically current research is focusing on a relaxed notion of the strict ACID database properties to allow for more flexible and fault tolerant architectures. Distributed cracking can exploit these new trends to potentially minimize the traffic creation caused by the self-organizing steps of cracking, i.e., data migration in a distributed environment.

8.10 Beyond the Horizon

Cracking sets a new query processing paradigm by reconsidering basic concepts of database design. This thesis showed that database cracking is possible and can lead to very promising results and the much desired self-organizing behavior without the need for external administration. It also opens a vast amount of interesting research challenges. The topics briefly described above are hard and rich research topics that if realized will be a significant step towards a truly functional cracking DBMS, opening new opportunities in complex and hard database scenarios where nowadays technology lacks the ability to perform.

Bibliography

Abadi, D. (2008). Query Execution in Column-Oriented Database Systems. *MIT PhD Thesis*.

Abadi, D., Madden, S. R., and Ferreira, M. C. (2006). Integrating compression and execution in column-oriented database systems. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*.

Abadi, D., Myers, D., DeWitt, D., and Madden, S. (2007). Materialization Strategies in a Column-Oriented DBMS. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Agrawal, S., Chaudhuri, S., Kollár, L., Marathe, A. P., arasayya, V. R., and Syamala, M. (2004). Database Tuning Advisor for Microsoft SQL Server. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Ailamaki, A., DeWitt, D., Hill, M., and Skounakis, M. (2001). Weaving Relations for Cache Performance. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Ailamaki, A., DeWitt, D., Hill, M., and Wood, D. (1999). DBMSs on modern processors: Where does time go? In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Bayer, R. and McCreight, E. M. (1972). Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189.

Bernstein, P. A., Giunchiglia, F., Kementsietsidis, A., Mylopoulos, J., Serafini, L., and Zaihrayeu, I. (2002). Data Management for Peer-to-Peer Computing: A Vision. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*.

Boncz, P., Manegold, S., and Kersten, M. (1999). Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Boncz, P., Wilschut, A., and Kersten, M. (1998). Flattening an Object Algebra

to Provide Performance. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*, pages 568–577, Orlando, FL, USA.

Boncz, P., Zukowski, M., and Nes, N. (2005). MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. of the Int'l Conf. on Innovative Database Systems Research (CIDR)*.

Bruno, N. and Chaudhuri, S. (2006). To Tune or not to Tune? A Lightweight Physical Design Alerter. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Chaudhuri, S. and Narasayya, V. (1997). An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Codd, A. (1970). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6).

Comer, D. (1979). The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137.

Copeland, G. and Khoshafian, S. (1985). A Decomposition Storage Model. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*.

DeWitt, D. J. and Gray, J. (1992). Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98.

Godfrey, P. and Gryz, J. (1997). Semantic query caching for hetereogeneous databases. In *Proc. of the Int'l Workshop on Knowledge Representation Meets Databases*.

Godfrey, P. and Gryz, J. (1999). Answering queries by semantic caches. In *Proc. of the Int'l. Workshop on Database and Expert Systems Application*.

Graefe, G. (1993). Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2).

Graefe, G. (2009). Fast loads and fast queries. In *DaWaK*.

Graefe, G. and Kuno, H. (2010a). Adaptive indexing for relational keys. In *Proc. of the Int'l. Workshop on Self-Managing Database Systems (SMDB)*.

Graefe, G. and Kuno, H. (2010b). Self-selecting, self-tuning, incrementally optimized indexes. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*.

Gribble, S., Halevy, A., Ives, Z., Rodrig, M., and Suciu, D. (2001). What Can Peer-to-Peer Do for Databases, and Vice Versa? In *Proceedings of the 4th International Workshop on the Web and Databases (WebDB)*.

Gupta, A. and Mumick, I. (1999). Materialized views: Techniques, implementations, and applications. *MIT Press*.

Hankins, R. A. and Patel, J. M. (2003). Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Harizopoulos, S., Liang, V., Abadi, D., and Madden, S. (2006). Performance Tradeoffs in Read-Optimized Databases. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S., and Stoica, I. (2003). Querying the internet with pier. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Idreos, S., Kaushik, R., Narasayya, V., and Ramamurthy, R. (2010a). "What-if" Compression Analysis. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Idreos, S., Kersten, M., and Manegold, S. (2007a). Database Cracking. In *Proc. of the Int'l Conf. on Innovative Database Systems Research (CIDR)*.

Idreos, S., Kersten, M., and Manegold, S. (2007b). Updating a Cracked Database. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*.

Idreos, S., Kersten, M., and Manegold, S. (2009). Self-organizing Tuple-reconstruction in Column-stores. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*.

Idreos, S., Kersten, M., and Manegold, S. (2010b). Adaptive Joins for Adaptive Kernels, Submitted for publication.

Idreos, S., Koubarakis, M., and Tryfonopoulos, C. (2004). P2p-diet: An extensible p2p service that unifies ad-hoc and continuous querying in super-peer networks. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*.

Idreos, S., Liarou, E., and Koubarakis, M. (2008). Continuous Multi-way Joins over Distributed Hash Tables. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*.

Idreos, S., Manegold, S., Graefe, G., and Kuno, H. (2010c). Adaptive Indexing, Submitted for publication.

Idreos, S., Tryfonopoulos, C., and Koubarakis, M. (2006). Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Ivanova, M., Kersten, M., Nes, N., and Goncalves, R. (2009). An architecture for recycling intermediates in a column-store. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*.

Kossmann, D. (2000). The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469.

Liarou, E., Idreos, S., and Koubarakis, M. (2006). Evaluating conjunctive triple pattern queries over large structured overlay networks. In *International Semantic Web Conference*.

Litwin, W., Neimat, M.-A., and Schneider, D. A. (1996). Lh* - a scalable, distributed data structure. *ACM Trans. Database Syst.*, 21(4):480–525.

Luo, G. (2007). Partial Materialized Views. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Manegold, S., Boncz, P., and Kersten, M. (2002). Optimizing Main-Memory Join On Modern Hardware. *IEEE Trans. on Knowledge and Data Eng.*, 14(4).

Manegold, S., Boncz, P. A., Nes, N., and Kersten, M. L. (2004). Cache-conscious radix-decluster projections. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Mehta, M. and DeWitt, D. (1997). Data placement in shared-nothing parallel database systems. *VLDB J.*, 6(1):53–72.

Mishra, P. and Eich, M. H. (1992). Join Processing in Relational Databases. *ACM Comput. Surv.*, 24(1).

MonetDB (2009). `http://monetdb.cwi.nl/`.

Papadomanolakis, S., Dash, D., and Ailamaki, A. (2007). Efficient Use of the Query Optimizer for Automated Database Design. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Qin, Y., Salem, K., and Goel, A. (2007). Towards Adaptive Costing of Database Access Methods. In *Proc. of the Int'l. Workshop on Self-Managing Database Systems (SMDB)*.

Ramamurthy, R., DeWitt, D., and Su, Q. (2002). A Case for Fractured Mirrors. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Schnaitter, K., Abiteboul, S., Milo, T., and Polyzotis, N. (2006). COLT: Continuous On-Line Database Tuning. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*.

Seshadri, P. and Swami, A. N. (1995). Generalized partial indexes. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Shao, M., Schindler, J., Schlosser, S. W., Ailamaki, A., and Ganger, G. R. (2004). Clotho: Decoupling Memory Page Layout from Storage Organization. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Shatdahl, A., Kant, C., and Naughton, J. (1994). Cache Conscious Algorithms for Relational Query Processing. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Stoica, I., Morris, R., L.-Nowell, D., Karger, D., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. (2003). Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32.

Stonebraker, M. (1989). The case for partial indexes. *SIGMOD Record*, 18(4):4–

11.

Stonebraker, M., Abadi, D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., E.O'Neil, O'Neil, P., Rasin, A., Tran, N., and Zdonik, S. (2005). C-Store: A Column Oriented DBMS. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Stonebraker, M., Aoki, P. M., Litwin, W., Pfeffer, A., Sah, A., Sidell, J., Staelin, C., and Yu, A. (1996). Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63.

TPC-H (2009). TPC Benchmark H. http://www.tpc.org/tpch/.

Tryfonopoulos, C., Idreos, S., and Koubarakis, M. (2005). Publish/subscribe functionality in IR environments using structured overlay networks. In *Proc. of the ACM SIGIR Int'l. Conf. on Information Retrieval*.

Valentin, G., Zuliani, M., Zilio, D. C., Lohman, G. M., and Skelley, A. (2000). DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Yu, C. T. and Chang, C. C. (1984). Distributed query processing. *ACM Comput. Surv.*, 16(4):399–433.

Zhou, J., Larson, P., Goldstein, J., and Ding, L. (2007). Dynamic Materialized Views. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Zhou, J. and Ross, K. A. (2003). A Multi-Resolution Block Storage Model for Database Design. In *Proc. of the Int'l. Database Engineering and Applications Symposium (IDEAS)*.

Zilio, D. C., Rao, J., Lightstone, S., Lohman, G. M., Storm, A., Garcia-Arellano, C., and Fadden, S. (2004). DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proc. of the Int'l. Conf. on Very Large Data Bases (VLDB)*.

Zukowski, M., Heman, S., Nes, N., and Boncz, P. (2006). Super-Scalar RAM-CPU Cache Compression. In *Proc. of the Int'l. Conf. on Database Engineering (ICDE)*.

Zukowski, M., Nes, N., and Boncz, P. A. (2008). DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proc. of the Int'l. Workshop on Data Management On New Hardware (DaMoN)*.

List of Figures

Summary

Indices are heavily used in database systems in order to achieve the ultimate query processing performance. It takes a lot of time to create an index and the system needs to reserve extra storage space to store the auxiliary data structure. When updates arrive, there is also the overhead of maintaining the index. This way, *which* indices to create and *when* to create them has been and still is one of the most important research topics over the last decades.

If the workload is known up-front or it can be predicted and if there is enough idle time to spare, then we can a priori create all necessary indices and exploit them when queries arrive. But what happens if we do not have this knowledge or idle time? Similarly, what happens if the workload changes often, suddenly and in an unpredictable way? Even if we can correctly analyze the current workload, it may well be that by the time we finish our analysis and create all necessary indices, the workload pattern has changed.

Here we argue that a database system should just be given the data and queries in a declarative way and the system should internally take care of finding not only the proper algorithms and query plans but also the proper physical design to match the workload and application needs. The goal is to remove the role of database administrators, leading to systems that can completely automatically self-tune and adapt even to dynamic environments. Database Cracking implements the first adaptive kernel that automatically adapts to the access patterns by selectively and adaptively optimizing the data set purely for the workload at hand. It continuously reorganizes input data on-the-fly as a side-effect of query processing using queries as an advice of how data should be stored. Everything happens within operator calls during query processing and brings knowledge to the system that future operators in future queries can exploit. Essentially, the necessary indices are built incrementally as the system gains more and more knowledge about the workload needs.

Samenvatting

Van oudsher spelen zogenaamde index-structuren een belangrijke rol binnen database systemen om de snelheid van query verwerking aanzienlijk te verbeteren. Echter, zulke index-structuren kosten veel tijd om aan te maken en nemen daarnaast veel ruimte in beslag. Verder zullen bij iedere wijziging binnen de database ook de aanwezige index-structuren moeten worden bijgewerkt, wat uiteraard ook kosten met zich meebrengt. Deze aanzienlijke snelheids verbetering, tegenover de extra toegevoegde kosten, heeft index-structuren een van de belangrijkste onderzoeksvragen gemaakt van de afgelopen decennia. Men heeft zich hier vooral toegelegd op de vragen *welke* index-structuren aan te maken, en *wanneer* dat te doen.

Het antwoord op wanneer en welke index-structuren aan te maken hangt af van de data die zich in het database systeem bevindt en van de queries die daarop worden uitgevoerd. Wanneer men aanneemt dat beide vooraf bekend zijn is het gemakkelijk om te bepalen wanneer welk type index-structuur nodig is voor een query, en wanneer een geschikt – bijvoorbeeld rustig – moment gevonden kan worden om het index-structuur aan te maken.

Echter, vooraf weten wat er gaat gebeuren is als in de toekomst kunnen kijken. In de werkelijkheid weet men zelden wat er in het database systeem gaat gebeuren, en dus zal men moeten voorspellen of gokken. Voorspellen dat er geen onregelmatigheden zullen ontstaan door veranderende omstandigheden. Gokken dat er nog op tijd een rustig moment zal komen. Hoe goed we ook in staat zijn met huidige technieken om de benodigdheden te herkennen van de queries die op dit moment op de huidige data in het database systeem worden uitgevoerd, tegen de tijd dat besloten is welke index-structuren moeten worden aangemaakt, kunnen deze alweer verouderd en niet optimaal zijn.

In dit proefschrift wordt de rol van het database systeem uitgebreid met de taak om niet meer achter de feiten aan te lopen, maar zelf een actief beleid te voeren ten opzichte van zijn data en de queries die daar op worden uitgevo-

erd. De aanmaak en onderhoud van index-structuren is niet langer de rol van een beheerder, maar van het database systeem die naast het vinden van de meest optimale manier om een query uit te voeren ook het gebruik van index-structuren over de data aanpast aan wat queries die uitgevoerd worden nodig hebben. *Database Cracking* is de eerste techniek voor een database systeem met aanpassend vermogen. De techniek past zich automatisch aan aan de benodigdheden van de queries die worden uitgevoerd door selectief aan een deel van een index-structuur te werken. Elke query heeft daardoor als neven-effect dat het index-structuur een stukje verfijnd wordt. Op deze manier wordt gaandeweg het index-structuur opgebouwd, gebaseerd op de benodigdheden van de uitgevoerde queries. De kosten hiervoor zijn op deze manier dus verspreid, terwijl de *welke* en *wanneer* vragen gebaseerd op de queries op de juiste manier worden beantwoord. Op deze manier is de aanmaak en onderhoud van index-structuren niet langer een taak voor de beheerder, maar past het database systeem zichzelf op de meest efficiente manier, waar nodig, aan.

Acknowledgments

There is only one author name in the cover of this book, but this is far from a correct reflection of how it was produced. This work is the result of endless joint brainstorming, hacking and writing sessions with my advisors Martin Kersten and Stefan Manegold. Their "obsession" to detail and innovation is what made this work possible.

Towards the end of this journey, Goetz Graefe and Harumi Kuno from HP Labs sparked new interest in this research. They extended the ideas and came up with new ones. A complete chapter of this thesis, Chapter 7, is based on joint work with Goetz and Harumi towards the vision of adaptive indexing.

Manolis Koubarakis plugged the research bug in me as my advisor in Technical University of Crete. At the time, I also started working closely with Christos Tryfonopoulos and Erietta Liarou, resulting in our first publications and conference travels. Developing a research character with Manolis, Christos and Erietta was and still is an invaluable experience that has greatly affected the way this work developed as well.

I would also like to thank the whole database architectures group at CWI. Other than a great place to work and the continuous feedback, especially from Marcin Zukowski and Peter Boncz, the development of the MonetDB system is what allowed this research to be applied and tested in a complete system.

I also spent several inspiring months doing three research internships in Microsoft Research in Redmond USA, in EPFL in Lausanne Switzerland and in IBM research in San Jose USA. For this great experience I would like to thank Surajit Chaudhuri, Vivek Narassaya, Ravi Rammamurty, Anastassia Ailamaki and Guy Lohman.

Traveling for conferences around the globe always felt a bit like going back home with so many members of the Greek database "mafia" present at any time and ready to provide feedback and motivation. I would especially like to thank Anastassia Ailamaki for all the small and big advises that have greatly affected

my research path and choices.

Finally, I would like to thank Hector Garcia-Molina, Surajit Chaudhuri, Arnold Smeulders, Peter Apers and Paul Klint for agreeing to be part of my PhD committee.

CURRICULUM VITAE

Education

10/2005-now	PhD candidate CWI Database group, Amsterdam, The Netherlands Supervised by Martin Kersten and Stefan Manegold
7/2003-9/2005	Master in Computer Engineering, 9.5/10 Department of Electronic and Computer Engineering Technical University of Crete Thesis: Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks, 10/10 Supervised by Manolis Koubarakis Committee: Stavros Christodoulakis and Euripidis Petrakis
9/1997-6/2003	Diploma in Electronic and Computer Engineering, 7.62/10 Department of Electronic and Computer Engineering Technical University of Crete Thesis: P2P-DIET: A query and notification service based on mobile agents for rapid implementation of peer-to-peer applications, 10/10 Supervised by Manolis Koubarakis Committee: Stavros Christodoulakis and Euripidis Petrakis

Employment & Academic Experience

10/2005-now	Junior researcher CWI, Database group Amsterdam, The Netherlands
7/2010-11/2010	Research intern IBM Almaden Research Center, DB group San Jose, California, USA
12/2009-4/2010	Research intern Data-Intensive Applications and Systems Laboratory, EPFL Lausanne, Switzerland
1/2009-4/2009	Research intern Microsoft Research, Data Management, Exploration and Mining group, Washington, Redmond, USA
8/2002-9/2005	Research assistant Intelligent Systems Laboratory, Technical University of Crete Crete, Greece
9/2003-9/2005	Teaching assistant Technical University of Crete, Greece Compilers (Autumn 2003, Autumn 2004) Distributed Systems (Spring 2004)

Publications

Refereed Conference Papers

(13) Stratos Idreos, Raghav Kaushik, Vivek Narasayya and Ravishankar Ramamurthy. Estimating the Compression Fraction of an Index using Sampling. In Proceedings of the 22nd IEEE International Conference in Data Engineering (**ICDE**), Long Beach, California, USA, March 2010

(12) Stratos Idreos, Martin Kersten and Stefan Manegold. Self-organizing Tuple Reconstruction In Column-stores. In Proceedings of the 29th ACM **SIGMOD** International Conference on Management of Data, Providence,

Rhode Island, USA, June 2009

(11) Erietta Liarou, Romulo Goncalves and Stratos Idreos. Exploiting the Power of Relational Databases for Efficient Stream Processing. In Proceedings of the 19th International Conference on Extending Database Technology (**EDBT**), Saint-Petersburg, Russia March 2009

(10) Stratos Idreos, Erietta Liarou and Manolis Koubarakis. Continuous Multi-Way Joins over Distributed Hash Tables. In Proceedings of the 11th International Conference on Extending Database Technology (**EDBT**), Nantes, France, March 2008

(9) Erietta Liarou, Stratos Idreos and Manolis Koubarakis. Continuous RDF Query Processing over DHTs. In Proceedings of the 6th International Semantic Web Conference (**ISWC**), Busan, Korea, November 2007

(8) Stratos Idreos, Martin Kersten and Stefan Manegold. Updating a Cracked Database. In Proceedings of the 27th ACM **SIGMOD** International Conference on Management of Data, Beijing, China, June 2007

(7) Stratos Idreos, Martin Kersten and Stefan Manegold. Database Cracking. In Proceedings of the 3rd International Conference on Innovative Data Systems Research (**CIDR**), Asilomar, California USA, January 2007

(6) Erietta Liarou, Stratos Idreos and Manolis Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In Proceedings of the 5th International Semantic Web Conference (**ISWC**), Athens, Georgia USA, November 2006

(5) Stratos Idreos, Christos Tryfonopoulos and Manolis Koubarakis. Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks. In Proceedings of the 22nd IEEE International Conference in Data Engineering (**ICDE**), Atlanta, Georgia USA, April 2006

(4) Christos Tryfonopoulos, Stratos Idreos and Manolis Koubarakis. Publish/Subscribe Functionality in IR Environments using Structured Overlay Networks. In Proceedings of the 28th Annual International ACM **SIGIR** Conference, Salvador, Brazil, August 2005

(3) Christos Tryfonopoulos, Stratos Idreos and Manolis Koubarakis. LibraRing: An Architecture for Future Digital Libraries Based on Structured

Overlay Networks. In Proceedings of the 9th European Conference on Research and Advanced Technology for Digital Libraries (**ECDL**), Vienna, Austria, September 2005. Best student paper award

(2) Paul-Alexandru Chirita, Stratos Idreos, Manolis Koubarakis and Wolfgang Nejdl. Publish/Subscribe for RDF-based P2P Networks. In Proceedings of the 1st European Semantic Web Conference (**ESWC**), Heraklion, Greece, May 2004

(1) Stratos Idreos and Manolis Koubarakis. P2P-DIET: Ad-hoc and Continuous Queries in Peer-to-Peer Networks using Mobile Agents. In Proceedings of the 3rd Hellenic Conference in Artificial Intelligence (**SETN**), Samos, Greece, May 2004

Refereed Demonstration Papers

(2) Stratos Idreos, Manolis Koubarakis and Christos Tryfonopoulos. P2P-DIET: An Extensible P2P Service that Unifies Ad-hoc and Continuous Querying in Super- peer Networks. In Proceedings of the ACM **SIGMOD** International Conference on Management of Data, Paris, France, June 2004

(1) Stratos Idreos, Manolis Koubarakis and Christos Tryfonopoulos. P2P-DIET: Ad-hoc and Continuous Queries in Super-peer Networks. In Proceedings of the 9th International Conference on Extending Database Technology (**EDBT**), Heraklion, Greece, March 2004

Refereed Journal Papers

(1) Manolis Koubarakis, Christos Tryfonopoulos, Stratos Idreos and Yannis Drougas. Selective Information Dissemination in P2P Networks: Problems and Solutions. ACM **SIGMOD Record**, Special issue on Peer-to-Peer Data Management, Karl Aberer (editor), 32(3), September 2003

Refereed Workshop Papers

(3) Erietta Liarou, Stratos Idreos and Manolis Koubarakis. Publish/Subscribe with RDF Data over Large Structured Overlay Networks. In Proceedings of the 3rd International Workshop on Databases, Information Systems and

Peer- to-Peer Computing (DBISP2P 2005) in conjunction with VLDB, Trondheim, Norway, August 2005

(2) Christos Tryfonopoulos, Stratos Idreos and Manolis Koubarakis. Publish/Subscribe Functionalities for Future Digital Libraries using Structured Overlay Networks. In Proceedings of the 8th International Workshop of the DELOS Network of Excellence on Digital Libraries on Future Digital Library Management Systems (System Architecture & Information Access), Schloss Dagstuhl, Germany, March 2005

(1) Stratos Idreos, Christos Tryfonopoulos, Manolis Koubarakis and Yannis Drougas. Query Processing in Super-Peer Networks with Languages Based on Information Retrieval: the P2P-DIET Approach. In Proceedings of the 1st Peer-to-peer Computing and Databases Workshop (P2P-DB) in conjunction with EDBT, Heraklion, Greece, March 2004

Book Chapters

(2) Zoi Kaoudi, Iris Miliaraki, Matoula Magiridou, Erietta Liarou, Stratos Idreos and Manolis Koubarakis. Semantic Grid Resource Discovery using DHTs in Atlas. In "Knowledge and Data Management in Grids", Talia Domenico, Bilas Angelos, and Dikaiakos Marios D.(editors), Springer, 2006.

(1) Paul-Alexandru Chirita, Stratos Idreos, Manolis Koubarakis and Wolfgang Nejdl. Designing Semantic Publish/Subscribe Networks Using Super-Peers. In "Semantic Web and Peer-to-Peer Systems". Steffen Staab and Heiner Stuckenschmidt(editors), Springer 2006

Reviewing

Reviewer	EDBT 2009 TKDE Internet Computing
External Reviewer	SIGMOD 2007, 2008, 2010 VLDB 2006, 2007, 2008, 2009 ICDE 2007, 2008, 2009, 2010 EDBT 2006 SOCC 2010

SIKS Dissertation Series

1998-1 Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM) Memory versus Search in Games

1998-5 E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999-1 Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR) Classification using decision trees and neural nets

1999-3 Don Beal (UM) The Nature of Minimax Search

1999-4 Jacques Penders (UM) The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU) Re-design of compositional systems

1999-7 David Spelt (UT) Verification support for object database design

1999-8 Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000-1 Frank Niessink (VU) Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE) Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UVA) Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.

2000-4 Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval

2000-6 Rogier van Eijk (UU) Programming Languages for Agent Communication

2000-7 Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR) Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI) Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI) Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI) Scalable Distributed Data Structures for Database Management

2001-1 Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA) Learning as problem solving

2001-4 Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU) Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002-01 Nico Lassing (VU) Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT) Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU) The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL) Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI) Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU) Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB) Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM) Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU) Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva) Processing XML in Database Systems

2002-13 Hongjing Wu (TUE) A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU) Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT) Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU) The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA) Understanding, Modeling, and Improving Main-Memory Database Performance

2003-01 Heiner Stuckenschmidt (VU) Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU) Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD) Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT) Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA) Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT) Development and specification of virtual environments

2003-07 Machiel Jansen (UvA) Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM) Repair Based Scheduling

2003-09 Rens Kortmann (UM) The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT) Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT) Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT) Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM) Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN) Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD) Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT) Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM) Learning Search Decisions

2004-01 Virginia Dignum (UU) A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT) Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UVA) Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR) Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD) The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek(UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politile gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU) For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UVA) Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU) Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT) Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU) Multi-Relational Data Mining

2004-16 Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM) Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005-01 Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM)) AI techniques for the game of Go

2005-03 Franc Grootjen (RUN) A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT) Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM) Adaptive Game AI

2005-07 Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR) Distributed Simulation in Industry

2005-13 Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU) Usability of XML Query Languages

2005-17 Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU) Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM) Situated Representation

2005-20 Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006-01 Samuil Angelov (TUE) Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU) Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UVA) The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU) Building Web Service Ontologies

2006-05 Cees Pierik (UU) Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU) Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT) XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM) Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU) Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT) Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU) Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU) Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU) Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU) CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU) Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU) User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UVA) Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU) Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT) Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN) Aptness on the Web

2006-22 Paul de Vrieze (RUN) Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU) Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU) Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU) Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlovic (UT) Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI) Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UVA) Focused Information Access using XML Element Retrieval

2007-01 Kees Leune (UvT) Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU) Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU) Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL) Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA) Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT) To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU) Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU) Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU) Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE) Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN) Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT) Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM) Context-Based Image Analysis

2007-15 Joyca Lacroix (UM) NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU) Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU) Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT) On the development an management of adaptive business collaborations

2007-19 David Levy (UM) Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU) Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU) Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT) Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE) Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramrez Camps (CWI) Structural Features in XML Retrieval

2007-25 Joost Schalken (VU) Empirical Investigations in Software Process Improvement

2008-01 Katalin Boer-Sorbn (EUR) Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU) On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA) Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT) Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT) Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN) On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU) Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU) Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU) The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT) Discourse oriented summarization

2008-11 Vera Kartseva (VU) Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN) A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UVA) Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT) Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT) The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 Henriette van Vugt (VU) Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD) Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM) Adaptive Active Vision

2008-19 Henning Rode (UT) From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UVA) Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer

met de overheid op de administratieve lasten van bedrijven.

2008-21 Krisztian Balog (UVA) People Search in the Enterprise

2008-22 Henk Koning (UU) Communication of IT-Architecture

2008-23 Stefan Visscher (UU) Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU) Using background knowledge in ontology matching

2008-25 Geert Jonker (UU) Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT) Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU) Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN) On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT) Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU) Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM) Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT) Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UVA) Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU) Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT) Dendritic morphologies: function shapes structure

2009-01 Rasa Jurgelenaite (RUN) Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU) Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT) A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN) Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN) Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU) Understanding Classification

2009-07 Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems

2009-10 Jan Wielemaker (UVA) Logic programming for knowledge-intensive interactive applications

2009-11 Alexander Boer (UVA) Legal Theory, Sources of Law & the Semantic Web

2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin) Operating Guidelines for Services

2009-13 Steven de Jong (UM) Fairness in Multi-Agent Systems

2009-14 Maksym Korotkiy (VU) From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)

2009-15 Rinke Hoekstra (UVA) Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 Fritz Reul (UvT) New Architectures in Computer Chess

2009-17 Laurens van der Maaten (UvT) Feature Extraction from Visual Data

2009-18 Fabian Groffen (CWI) Armada, An Evolving Database System

2009-19 Valentin Robu (CWI) Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets

2009-20 Bob van der Vecht (UU) Adjustable Autonomy: Controling Influences on Decision Making

2009-21 Stijn Vanderlooy (UM) Ranking and Reliable Classification

2009-22 Pavel Serdyukov (UT) Search For Expertise: Going beyond direct evidence

2009-23 Peter Hofgesang (VU) Modelling Web Usage in a Changing Environment

2009-24 Annerieke Heuvelink (VUA) Cognitive Models for Training Simulations

2009-25 Alex van Ballegooij (CWI) "RAM: Array Database Management through Relational

Mapping"

2009-26 Fernando Koch (UU) An Agent-Based Model for the Development of Intelligent Mobile Services

2009-27 Christian Glahn (OU) Contextual Support of social Engagement and Reflection on the Web

2009-28 Sander Evers (UT) Sensor Data Management with Probabilistic Models

2009-29 Stanislav Pokraev (UT) Model-Driven Semantic Integration of Service-Oriented Applications

2009-30 Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage

2009-31 Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text

2009-32 Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors

2009-33 Khiet Truong (UT) How Does Real Affect Affect Affect Recognition In Speech?

2009-34 Inge van de Weerd (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach

2009-35 Wouter Koelewijn (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling

2009-36 Marco Kalz (OUN) Placement Support for Learners in Learning Networks

2009-37 Hendrik Drachsler (OUN) Navigation Support for Learners in Informal Learning Networks

2009-38 Riina Vuorikari (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin) Service Substitution – A Behavioral Approach Based on Petri Nets

2009-40 Stephan Raaijmakers (UvT) Multinomial Language Learning: Investigations into the Geometry of Language

2009-41 Igor Berezhnyy (UvT) Digital Analysis of Paintings

2009-42 Toine Bogers (UvT) Recommender Systems for Social Bookmarking

2009-43 Virginia Nunes Leal Franqueira (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients

2009-44 Roberto Santana Tapia (UT) Assessing Business-IT Alignment in Networked Organizations

2009-45 Jilles Vreeken (UU) Making Pattern Mining Useful

2009-46 Loredana Afanasiev (UvA) Querying XML: Benchmarks and Recursion

2010-01 Matthijs van Leeuwen (UU) Patterns that Matter

2010-02 Ingo Wassink (UT) Work flows in Life Science

2010-03 Joost Geurts (CWI) A Document Engineering Model and Processing Framework for Multimedia documents

2010-04 Olga Kulyk (UT) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments

2010-05 Claudia Hauff (UT) Predicting the Effectiveness of Queries and Retrieval Systems

2010-06 Sander Bakkes (UvT) Rapid Adaptation of Video Game AI

2010-07 Wim Fikkert (UT) A Gesture interaction at a Distance

2010-08 Krzysztof Siewicz (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments

2010-09 Hugo Kielman (UL) A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging

2010-10 Rebecca Ong (UL) Mobile Communication and Protection of Children

2010-11 Adriaan Ter Mors (TUD) The world according to MARP: Multi-Agent Route Planning

2010-12 Susan van den Braak (UU) Sensemaking software for crime analysis

2010-13 Gianluigi Folino (RUN) High Performance Data Mining using Bio-inspired techniques

2010-14 Sander van Splunter (VU) Automated Web Service Reconfiguration

2010-15 Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models

2010-16 Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice

2010-17 Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

2010-18 Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation

2010-19 Henriette Cramer (UvA) People's Responses to Autonomous and Adaptive Systems

2010-20 Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative

2010-21 Harold van Heerde (UT) Privacy-aware data management by means of data degradation

2010-22 Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data

2010-23 Bas Steunebrink (UU) The Logical Structure of Emotions

2010-24 Dmytro Tykhonov Designing Generic and Efficient Negotiation Strategies

2010-25 Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective

2010-26 Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines

2010-27 Marten Voulon (UL) Automatisch contracteren

2010-28 Arne Koopman (UU) Characteristic Relational Patterns

2010-29 Stratos Idreos (CWI) Database Cracking: Towards Auto-tuning Database Kernels