Query processing of pre-partitioned data using Sandwich Operators

Stephan Baumann[1], Peter Boncz[2], and Kai-Uwe Sattler[1]

[1] Ilmenau University of Technology, Ilmenau, Germany,
first.last@tu-ilmenau.de,
[2] Centrum for Wiskunde, Amsterdam, Netherlands
boncz@cwi.nl

Abstract. In this paper we present the "Sandwich Operators", an elegant approach to exploit pre-sorting or pre-grouping from clustered storage schemes in operators such as `Aggregation/Grouping`, `HashJoin`, and `Sort` of a database management system. Thereby, each of these operator types is "sandwiched" by two new operators, namely `PartitionSplit` and `PartitionRestart`. `PartitionSplit` splits the input relation into its smaller independent groups on which the sandwiched operator is executed. After a group is processed `PartitionRestart` is used to trigger the execution on the following group. Executing one of these operator types with the help of the Sandwich Operators introduces minimal overhead and does not penalty performance of the sandwiched operator as its implementation remains unchanged. On the contrary, we show that sandwiched execution of an operator results in lower memory consumption and faster execution time. `PartitionSplit` and `PartitionRestart` replace special implementations of partitioned versions of these operator. Sandwich Operators also turn blocking operators in streaming operators, resulting in faster response times for the first query results.

Key words: ordering, partitioned data, query processing

1 Introduction

Today, data warehouses for various reporting and analytical tasks are typically characterized by huge data volumes and a desire for interactive query response times. Over the last few years, many different techniques have been developed to address these challenges. Examples are techniques to reduce I/O by using columnar data organization schemes and/or data compression, to avoid the I/O bottleneck by keeping the data in memory (in-memory processing), and to exploit the computing power of modern hardware by using parallelization, vectorization as well as cache-conscious techniques.

Orthogonal to such improvements in raw query execution performance are existing techniques like table partitioning, table ordering and table clustering which are already found in many RDBMS products. Most of the commercial and open source databases systems support some kind of (horizontal) partitioning of tables and indexes [15, 9, 2]. Such partitioning is not only useful to support parallel access and to allow the query planner to prune unneeded data, but provides basically a grouping of tuples. Another related technique is clustering which also stores logically related data together. Examples are Multidimensional Clustering (MDC) in IBM DB2 [11] or partitioned B-trees [3], which are defined by distinct values in an artificial leading key column instead of a definition in the catalog. In the world of column stores, finally, storage in (multiple) ordered projections also provides a way to access data grouped by order key (range).

Though partitioning is based on a physical data organization whereas sorting and clustering are more on a logical level within the same data structure, all these techniques share a common basic concept: grouping of tuples based on some criteria like (combinations of) attribute values.

Clustering and ordering are currently most considered for indexing and exploited for accelerating selections: selection predicates on the grouping keys (or correlated with these) typically can avoid scanning large part of the data. Table partitioning also leverages this through *partition pruning*: a query planner will avoid to read data from table partitions whose data would always be excluded by a selection predicate. For joins, it holds that these are exploited primarily in table partitioning, if the partitioning key was fetched over a foreign key. In this case, for evaluating that foreign key join, only the matching partitions need to be joined. Partitioning is typically implemented by generating separate scans of different partitions, and partially replicating the query plan for each partition, which leads to a query plan blow-up. In [4] a technique is investigated to counter the ill effects of such blow-up on query optimization complexity.

In this paper we introduce a generalization of the grouping principle that underlies table partitioning, clustering and ordering, and that allows to elegantly exploit such grouping in query plans without causing any query plan blow-up. The basic idea is to not only make join operators, but also aggregation and sort operators, exploit relevant grouping present in the stream of the tuples they process. If this grouping is determined by the join, aggregation or sort key, the operator can already generate all results so far as soon as its finishes with one group, and the next group starts. Additionally, memory resources held in internal hash tables can already be released, reducing resource consumption.

The elegance of our approach is found in two key aspects. The first is the purely logical approach that treats grouping as a *tuple ordering property* captured in a synthetic `_groupID_` column, rather than physical groups, produced by separate operators. This avoids the plan blow-up problem altogether (additionally makes grouping naturally fit query optimization frameworks that exploit interesting orderings – though for space reasons, query optimization beyond this paper's scope). The second elegant aspect are the *Sandwich Operators* we propose, that allow to exploit ordering in join, aggregation and sort operators without need for creating specialized grouped variant implementations. This approach *sandwiches* the grouped operator between `PartitionSplit` and `PartitionRestart` operators that we introduce; and exploit the iterator model with some sideways information passing between `PartitionSplit` and `PartitionRestart`.

The remainder of this paper is structured as follows: After introducing preliminaries and basic notions in Sect. 2, we discuss the opportunities and use cases of the sandwiching scheme in Sect. 3. The new query operators implementing this sandwiching scheme are presented in Sect. 4. We implemented sandwich operators in a modified version of Vectorwise [5, 17][1]. However, the general approach can easily be adopted by other systems. In Sect. 5 we discuss necessary steps and requirements and give an example. Our experiments on microbenchmarks and all 22 TPC-H queries in Sect. 6 show advantages in speed, reduced memory consumption and negligible overhead addressing the challenges of realtime data warehousing. Finally, we conclude in Sect. 8 and point out future work.

2 Preliminaries

For easier understanding we follow two definitions introduced in [16]. The first defines a physical Relation with the help of the total order $\rhd_R$

Definition 1 (Physical Relation). *A* physical relation *R is a sequence of n tuples* $t_1 \rhd_R t_2 \rhd_R \ldots \rhd_R t_n$*, such that "*$t_i \rhd_R t_j$*" holds for records* t_i *and* t_j*, if* t_i *immediately precedes* t_j*,* $i, j \in \{1, \ldots, n\}$*.*

In the following we will use physical relation and tuple stream or input stream interchangeable. A second definition only given informally in [16] is that of an order property, which will be sufficient for our purposes here.

Definition 2 (Order Property). *For a subset* $\{A_1, \ldots, A_n\}$ *of Attributes of a Relation R and* $\alpha_i \in \{O, G\}$ *(*$\alpha_i = O$ *defining an ordering,* $\alpha_i = G$ *defining a grouping), the sequence*

$$A_1^{\alpha_1} \rightarrow A_2^{\alpha_2} \rightarrow \ldots \rightarrow A_n^{\alpha_n}$$

is an attribute sequence that defines an order property for R, such that the major ordering/grouping is $A_1^{\alpha_1}$*, the secondary ordering is* $A_2^{\alpha_2}$ *and so on.*

Here, ordering (for simplicity only ascending) of an attribute A_i means that tuples of R will follow the order of the values of column A_i. Grouping of an attribute A_j is not as strong and only means that tuples with the same value for attribute A_j will be grouped together in R, but tuples may not be ordered according to values of A_j. For further reading we refer to [16].

Definition 3 (Group Identifier). *A group identifier* `_groupID_` *is an additional implicit attribute to a relation R, representing the order property* $A_1^{\alpha_1} \rightarrow A_2^{\alpha_2} \rightarrow \ldots \rightarrow A_n^{\alpha_n}$ *of R. The values of attribute* `_groupID_` *are the result of a bijective mapping function* $f : (A_1, \ldots, A_n) \rightarrow \{1, \ldots, m\}$*, where a* t_1*.*`_groupID_` *is smaller than* t_2*.*`_groupID_`*, if and only if* t_1 *precedes* t_2 *in R.*

This means, each value of `_groupID_` represents a value combination of the attributes present in the order property. In addition, `_groupID_` is reconstructable from a value combination of these attributes. Explicitly, each single occurrence of a value of an attribute is reconstructable from `_groupID_`.

[1] Vectorwise is a further development of X100 [17].

3 Motivation

Various table storage schemes result in in a form of data organization, where a subset of all table attributes determine an ordering or grouping of the stored data. Examples of these schemes are amongst others MDC [11] or ADC [10] or MDAM [6], where data is organized by a number of dimensions and can be retrieved in different orders. Additionally, column stores sometimes stored data in (multiple, overlapping) sorted projections [14]. These methods have in common, that data is not physically partitioned but has a physical ordering or grouping defined over one or multiple attributes that can be exploited during query execution. Any index scan results in a relation that contains valuable information about an ordering or grouping already present in the tuple stream. Our sandwich approach is based on having one of these forms of data organization and is designed to exploit such pre-ordering or pre-grouping. However, even systems implementing physical partitioning over one or more attributes provide the same valuable information when multiple partitions are combined into a single stream. Assuming a tuple stream that has a certain order or suborder defined over a set of attributes, we can find standard operators and show potential for optimization.

3.1 `Aggregation/Grouping`

In case of hash-based `Aggregation/Grouping`, if any subset G_s of the GROUPBY keys determines a sub-sequence $A_1^{\alpha_1} \rightarrow \ldots \rightarrow A_k^{\alpha_k}$ of the order $A_1^{\alpha_1} \rightarrow \ldots \rightarrow A_n^{\alpha_n}$ of the input tuple stream, $k \leq n$, we can flush the operator's hash table and emit results as soon as the entire group - each group is defined by `_groupID_` - is processed. Effectively, we execute the `Aggregation/Grouping` as a sequence of `Aggregation/Grouping` operators, each of which operating on only one group of data. This results in the `Aggregation/Grouping` behaving more like a non-blocking, pipelined operator, emitting results on a per group basis. Additionally, memory consumption should drop down, as the hash table only needs to be built on a subset of all keys. This may cause `Aggregation/Grouping` to no longer spill to disk, or its hash-table may become TLB or CPU cache resident. And as a side effect from the reduced memory consumption we should get an improved execution time of the `Aggregation/Grouping`.

3.2 `Sort`

If a prefix $A_1^O \rightarrow \ldots \rightarrow A_k^O$ of the input relation's order $A_1^O \rightarrow \ldots \rightarrow A_n^O$ represents the same ordering as a prefix $B_1^O \rightarrow \ldots \rightarrow B_l^O$ of the requested sort order $B_1^O \rightarrow \ldots \rightarrow B_m^O$, then the tuple stream is already pre-sorted at no cost for $B_1, \ldots, B_l$ and only needs to be sorted on the remaining minor sort keys $B_{l+1}, \ldots, B_m$. This again results in executing `Sort` as a sequence of `Sort`s, each working only on a fraction of the data. The benefits of a grouped `Sort` should be similar to grouped `Aggregation/Grouping`, but additionally, as data is only sorted in small groups, the computational complexity also decreases.

3.3 HashJoin

In case of any kind of HashJoin - this also includes Semi-, Anti- and Outer-Hash-Joins - if a subset K_s of the join keys determines a prefix $A_1^{\alpha_1} \to \ldots \to A_k^{\alpha_k}$ of the order $A_1^{\alpha_1} \to \ldots \to A_n^{\alpha_n}$ of one input tuple stream, $k \leq n$, and K_s also determines a prefix $B_1^{\alpha_1} \to \ldots \to B_k^{\alpha_k}$ of the order $B_1^{\alpha_1} \to \ldots \to B_m^{\alpha_m}$ of the second input tuple stream, $k \leq m$, we can transform the task into multiple Hash-Joins, where the grouping is already present, and only matching groups induced by K_s are joined. Similar to sandwiched Aggregation/Grouping, this should result in smaller hash tables and, thus, less memory consumption for the build phase and, as a consequence from the reduced memory, better cache awareness for the build and probe phases, as well as better pipelining performance from the grouped processing. If, in addition, in both cases $\alpha_i = O, 1 \leq i \leq k$, i.e. there are only orderings involved and _groupID_ is a strictly ascending column, we can use merge techniques between the groups and skip the execution of the HashJoin for complete groups if there is no matching _groupID_.

4 Sandwich Operators

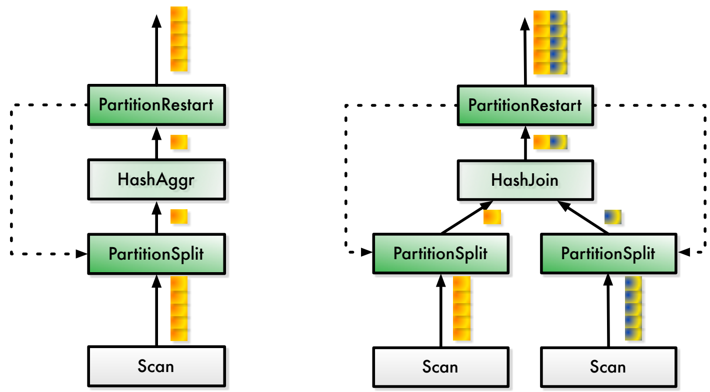
In this section we introduce two new *Sandwich* operators PartitionSplit and PartitionRestart that enable the use of (almost) unmodified existing Sort, Aggregation/Grouping and HashJoin operators to exploit partial pre-ordering of their input streams. Let this partial order be represented by an extra column called _groupID_ as introduced in 3.

The following algorithms illustrate our implementation in Vectorwise. Note that Vectorwise realizes vectorized processing of data [17], where an operator handles a vector of data at a time instead of just a tuple at a time. This enables further optimizations but the core ideas of our algorithms are transferable to tuple-at-a-time pipelining systems.

4.1 Sandwich Algorithms

Instead of implementing a partitioned variant of each physical Sort, Hash-Join and Aggregation/Grouping operator, we devise a *split* and *restart* approach where the "sandwiched" operator is tricked into believing that the end-of-group is end-of-stream, but after performing its epilogue action (e.g. Aggregation/Grouping emitted all result tuples), the operator is restarted to perform more work on the next group, reusing already allocated data structures.

For this purpose we added two new query operators PartitionSplit(*stream*, *_groupID_-col*) and PartitionRestart(*stream*). The basic idea of these operators is illustrated in Fig.1 for an unary operator, HashAggr(1(a)), and a binary operator, HashJoin (1(b)). The PartitionSplit operator is inserted below Sort, HashJoin or Aggregation/Grouping and the PartitionRestart on top. PartitionSplit is used to detect group boundaries of the input stream using attribute _groupID_ and to break it up into chunks at the detected boundaries.



(a) Example Sandwich Aggregation (b) Example Sandwich HashJoin

Fig. 1. Sandwich Operators with sideways information passing

`PartitionRestart` controls the sandwiched operators restart after it finished producing tuples for a group, passes on the result tuples to the next operator and notifies its corresponding `PartitionSplit` operator(s), that the sandwiched operator is ready to process the next group. Note that this communication between `PartitionRestart` and `PartitionSplit` is a form of sideways information passing, for which `PartitionRestart` has to know the corresponding `PartitionSplit` operator(s) in the plan. These are determined during query initialization, and typically are its grandchildren.

For both operators we outline their `Next()` methods. We also explain how Sandwich Operators are used to process the pre-grouped data of a `HashJoin` in a merge like fashion, where groups are skipped if group identifiers do not match.

PartitionSplit() controls the amount of tuples that are passed to the sandwiched operator. When a group boundary is detected, `PartitionSplit` stops producing tuples and signals the sandwiched operator *end-of-input* while waiting for its corresponding `PartitionRestart` to signal to produce tuples again.

`PartitionSplit` uses the following member variables:

- `run` - state of the operator; RUN for producing, STOP at the end of a group, END when finished with all groups.
- `net` - current potion in the current vector
- `redo` - signal to produce last group once more
- `veclen` - length of current vector
- `vec` - current vector
- `grp` - current group identifier

This means the `PartitionSplit.Next()` method, as shown in Algorithm 1, just forwards tuples until a group border is detected. In line 9 `SkipVal(this.grp)` finds the number of tuples of the remaining range `[this.nxt,this.veclen]` of vector `this.grp` that belong to the same group, i.e. it finds the position where the `_groupID_` changes. On the next invocation after such a group border has been reached and all its tuples have been passed, the method has set `run=STOP`

```
   // initially: run=RUN nxt=veclen=redo=grp=0
1  if this.run ≠ RUN then return 0;
2  if ¬this.redo then
3      if this.nxt = this.veclen then
4          this.veclen ← Child.Next(); // get new tuples
5          this.nxt ← 0;
6          if this.veclen = 0 then
7              this.run ←END; // real end-of-stream
8              return 0;
9      n←this.vec[this.nxt..this.veclen].SkipVal(this.grp);
10     this.nxt ← this.nxt + n;
11     if this.nxt < this.veclen then
12         this.grp ← this.vec [this.nxt ]._groupID_; // advance to next group ID
13         this.run ← STOP; // next group in sight
14 else
15     n ← this.redo;
16     this.redo ← 0 // see SkipGrp
17 return n; // return vector of n tuples
```

Algorithm 1: `PartitionSplit.Next()`

(line 13) and returns 0 (line 1), signaling (deceivingly) end-of-stream to the parent operator. This will lead an `Aggregation/Grouping` to emit aggregate result tuples of the, to this point, aggregated values, i.e. aggregates over the current group, after which it will pass 0 to its parent, in general `PartitionRestart`. In a similar way `Sort` will produce a sorted stream of the current group and `HashJoin` will either switch from building to probing or produce result tuples for the current group, depending on which `PartitionSplit` sent the end-of-stream signal. When `PartitionSplit.Next()` is called after it had previously stopped, it first checks if there are still tuples left in the current vector (line 3) and, if needed, fetches a new vector (line 4) or switches to `run=END` (line 7) if the final vector was processed.

PartitionRestart() controls the restart of the sandwiched operator and its associated `PartitionSplit`(s). In addition it applies the merge techniques in case of a sandwiched `HashJoin`.

`PartitionRestart` has the following member variables:

- `Child` - the operator below in the operator tree, usually the sandwiched operator
- `lSplit` - the corresponding (left, in case of a binary sandwich) `PartitionSplit`
- `rSplit` - in right `PartitionSplit` in case of a binary sandwiched operator

The `PartitionRestart.Next()` method (Algorithm 2) also just passes on tuples (line 9), until it receives an end-of-stream signal from its `Child` (line 3). For a unary operator it de-blocks its corresponding `PartitionSplit` if it was `STOP`ped (lines 5,6). For a binary operator it calls `PartitionRestart.GroupMergeNext()` (line 7) which handles the de-blocking of the two `PartitionSplit` operators in

```
1 n ← 0;
2 while n = 0 do
3     if (n ← this.Child.Next()) = 0 then
4         if IsUnarySandwich(this) then
5             if this.lSplit.run = END then break;
6             this.lSplit.run ← RUN; // deblock Split
7         else if ¬GroupMergeNext() then break;
8         this.Child.Restart(); // e.g., flush Aggregation/Grouping hashtable
9 return n; // return vector of n tuples
```

Algorithm 2: `PartitionRestart.Next()`

this case. Finally, it restarts the sandwiched child in line 8. If the `PartitionSplit` operators do not have any more input data for the sandwiched operator, then the `while` loop is exited in either line 5 or line 7 and 0 is returned, signaling end of stream to the operators further up in the tree.

```
while this.lSplit.grp ≠ this.rSplit.grp do
    if this.lSplit.grp > this.rSplit.grp then
        if ¬this.rSplit.SkipGrp(this.lSplit.grp) then break;
    else if this.lSplit.grp < this.rSplit.grp then
        if ¬this.lSplit.SkipGrp(this.rSplit.grp) then break;
return this.lSplit.run ≠ END and this.rSplit.run ≠ END;
```

Algorithm 3: `PartitionRestart.GroupMergeNext()`

The group based merge join is implemented in `PartitionRestart.GroupMergeNext()` (see Algorithm 3) using a merge-join between the `PartitionSplit` operators to match groups from both input streams on its _groupID_ values. Of course it is necessary here, that _groupID_ is not only an identifier but sorted ascending or descending on both sides. It is given here for `Inner-HashJoin`: for `Outer-` and `Anti-HashJoins` it should return matching success even if one of the sides does not match (i.e. an empty group).

```
n ← 0;
while this.grp < grp do
    if this.run ≠ END then
        this.run ←RUN; // force progress
        n ← PartitionSplit.Next();
    if this.run = END then return FALSE;
    if this.vec [this.veclen − 1]._groupID_ < grp then
        this.nxt ← this.veclen; // vector shortcut to skip search in Next()
this.redo ← n;
this.run ← RUN // Next() returns vector again
return TRUE;
```

Algorithm 4: `PartitionSplit.SkipGrp(grp)`

It uses `PartitionSplit.SkipGrp` (Algorithm 4) to advance over groups as long as the current `_groupID_` is still smaller than the target `_groupID_`. In turn `PartitionSplit.SkipGrp` calls `PartitionSplit.Next()` (line 5) to find the next `_groupID_` (Algorithm 1, line 12). In lines 7-8 a shortcut is used to avoid skipping over every distinct `_groupID_` in the vector (setting this.nxt to the vector length will trigger the call for the next vector in `PartitionSplit.Next()`, line 3-4). The `redo` variable used in both methods is needed, as the sandwiched operator's `Next()` call needs to receive the last tuple vector once more.

Recall, that in our test implementation in Vectorwise these methods manipulate vectors rather than individual tuples, which reduces interpretation overhead and offers algorithmic optimization opportunities. For instance, the `SkipVal()` routine (not shown) uses binary search inside the vector to find the next group boundary, hence group finding cost is sub-linear. Another example is the vector shortcut in line 7 of Algorithm 4, where an entire vector gets skipped in `GroupMergeNext()` based on one comparison – checking if the last value in the vector is still too low.

We extended the (vectorized) open(), next(), close() operator API in Vectorwise with a restart() method to enable operators to run in sandwich – note that many existing database systems already have such a method (used e.g. in executing non-flattened nested query plans). This restart() method has the task to bring an operator in its initial state; for hash-based operators it typically flushes the hash table. A workaround could be to re-initialize which may result in somewhat slower performance.

5 Application of Sandwich Operators

In order to introduce sandwich operators into query plans, the system needs to be able to generate and detect operator sandwiching opportunities.

5.1 Order Tracking and Analysis.

Table partitioning, indexing, clustering and ordering schemes, can efficiently produce sorted or grouped tuple streams in a scan. Though these various approaches, and various systems implementing them, handle this in varied ways, conceptually (and often practically) it is easy to add a proper `_groupID_` column to such a tuple stream. Note that we make little assumptions on the shape of this `_groupID_` column. It does not need to be a simple integer, since our `PartitionSplit` and `PartitionRestart` operators can in fact trivially work with multi-column group keys as well. In the following, we abstract this into a scan called `GIDscan`, that a) adds some `_groupID_` column and b) produces a stream ordered on `_groupID_`.

Such ordering/grouping on `_groupID_` from a `GIDscan` will propagate through the query plan as described in [16]. In our system Vectorwise, the operators `Project`, `Select` and the left (outer) side of joins preserve order and were used for order propagation.

Formally, the original ordering or grouping attributes functionally determine the _groupID_ column. If the optimizer has metadata about functional dependencies between combinations of attributes, it will be able to infer that other groups of attributes also determine the _groupID_ column. This order and grouping tracking and functional dependency analysis during query optimization should go hand-in-hand with tracking of foreign key joins in the query plan. The order and grouping tracking allows to identify whether aggregation and sort keys determine a _groupID_, providing a sandwich opportunity. The additional foreign key tracking in combination with this, allows a query optimizer to detect that the join keys on both sides on the join are determined by matching _groupID_ columns (groups with the same boundaries), such that join results can only come from matching _groupID_ groups on both sides of a join. This allows to identify sandwiching opportunities for joins.

5.2 Query Optimization

Sandwiched query operators consume much less memory and run faster due to better cache locality but also because its reduced memory consumption will typically eliminate the need for disk spilling, if there was one. Therefore, the query optimizer, and in particular its cost model, should be made aware of the cost of sandwiched operators. Note, that estimating the cost of the PartitionSplit and PartitionRestart operators is not the problem here, as they only bring linear (but low) CPU cost in terms of the amount of tuples that stream through them. If fact, thanks to the vectorized optimizations that we outlined, their cost is actually sub-linear: (i) finding group boundaries in PartitionSplit uses binary search, and (ii) the merge join between groups in PartitionRestart typically only looks at the first and last vector values, thanks to the skip optimization. Therefore, our cost model just ignores the cost of these two operators, and focuses on adapting the cost of the sandwiched aggregation, join and sort operators. The cost model extensions are quite simple and are based on the number of groups γ_{rel} present in an input relation rel. Sort costs decrease from $O(N \cdot log(N))$ to $O(N \cdot log(N/\gamma_{rel}))$. For hash-based aggregation and HashJoin, one simply reduces the hash table size fed into any existing cost model (e.g. [7]) by factor γ_{rel}.

As for the bigger picture in sandwiched query optimization, we note that in a multi-dimensional setup such as MDC or any partitioning, indexing, clustering or ordering scheme with a multi-column key, it may be possible to efficiently generate tuples in *many orders*: potentially for any ordering of any a subset of these keys. The potential to generate such different ordered tuple streams leads to different, and sometimes conflicting sandwiching opportunities higher up in the plan. Due to space restrictions, the question how to choose the best orderings is beyond the scope of this paper, but we can note here that our solution seamlessly fits into the well known concept of interesting order optimization [13], and on which we will report in a subsequent paper.

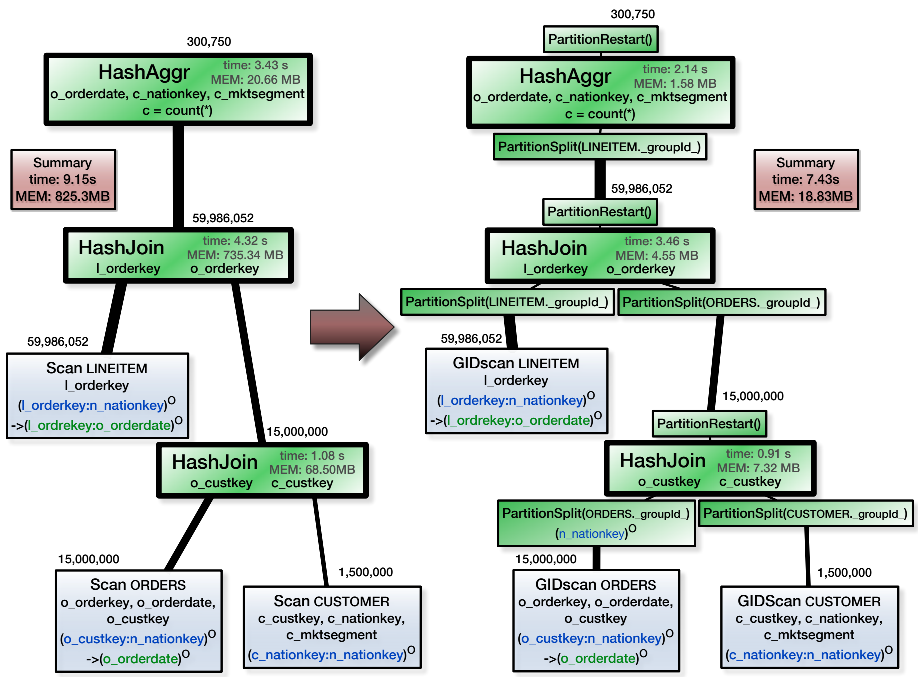


Fig. 2. Example query with and without sandwich operators.

5.3 An Example

In Figure 2 we demonstrate the use of sandwich operators in the following query on the TPC-H dataset:

```
SELECT o_orderdate, c_nationkey, count(*)
FROM CUSTOMER, ORDERS, LINEITEM
WHERE c_custkey=o_custkey
  AND o_orderkey=l_orderkey
GROUP BY o_orderdate, c_nationkey, city(c_address)
```

This query is a simple version of counting the number of lineitems per market segment of each nation and date. The operators of interest are annotated with overall memory consumption and execution time, explaining the overall gain as summarized in the two summary boxes.

Assume the tables to be organized according to the following order properties:

CUSTOMER : $\texttt{c\_nationkey}^{O}$
ORDERS : $\texttt{[o\_custkey.c\_custkey].c\_nationkey}^{O} \rightarrow \texttt{o\_orderdate}^{O}$
LINEITEM : $\texttt{[l\_orderkey.o\_orderkey].[o\_custkey.c\_custkey].n\_nationkey}^{O}$
$\rightarrow \texttt{[l\_orderkey.o\_orderkey].o\_orderdate}^{O}$

where $[A_1, A_2]$ denote a foreign key relationship between two tables, i.e. [o_custkey.c_custkey].c_nationkey^O means that ORDERS is major sorted according to the customer nations.

As there is no information about the order of ORDERS or LINEITEM inside the nation/date groups , the original plan is still a hash based plan. Same holds for the ordering of CUSTOMER and ORDERS and ordering information about custkey.

However, as we have ordering properties of the tuple streams we can perform the following sandwich optimizations:

- HashJoin(ORDERS, CUSTOMER): Both join keys determine the ordering on n_nationkey. Thus, the grouping on CUSTOMER can fully be exploited. Note, that ORDERS has a more detailed grouping, i.e. in addition to n_nationkey also o_orderdate, and for the split only the grouping on n_nationkey is taken into account. This is possible as we constructed _groupID_ in a way that enabled the extraction of major orderings (see Sect. 2). In order to sandwich the HashJoin, PartitionRestart is inserted on top and one PartitionSplit per child is inserted on top of each input stream. PartitionSplit for the ORDERS stream needs to be provided with an extraction function of only the n_nationkey ordering. This results in 9x reduced memory consumption and 16% speedup.
- HashJoin(LINEITEM, ORDERS): Here, both join keys determine the full ordering as given be the order properties, so the sandwich can is over the complete pre-ordering. PartitionRestart and PartitionSplit are inserted similar to the first case. As this sandwich operation exploits even more bits, the memory reduction is even more significant (161x), also the speedup with 20% is higher.
- HashAggr(o_orderdate, c_nationkey, c_city): Two of three grouping keys, i.e. o_orderkey and c_nationkey not only determine the ordering of the input stream but also are determined by LINEITEM._groupID_. That means the aggregation can be sandwiched using this pre-ordering and is only performed on a per city basis. For the HashAggr, again, a PartitionRestart is inserted on top and a PartitionSplit on LINEITEM._groupID_ is inserted on top of its input stream. Reducing the HashAggr to a per city basis accelerates the operator by 38% and reduces memory needs 13-times.

Note, that in all cases, input data already arrives in a cache friendly order, i.e. the input streams are grouped in similar ways. This means that for example in the ORDERS-CUSTOMER join customers as well as orders are already grouped by nation. This already results in locality for the HashJoin itself, an effect that is again amplified by the sandwich operators as the hash table size is shrunk.

6 Evaluation of Sandwich Operators

We evaluated on an Intel Xeon E5505 2.00 GHz with 16GB main memory, a standard 1TB WD Caviar Black hard drive for the operating system, a 64 bit Debian Linux with kernel version 2.6.32. The system has 4 cores with 32KB L1, 256KB L2 and 4096KB L3 cache per core. Databases were stored on a RAID0

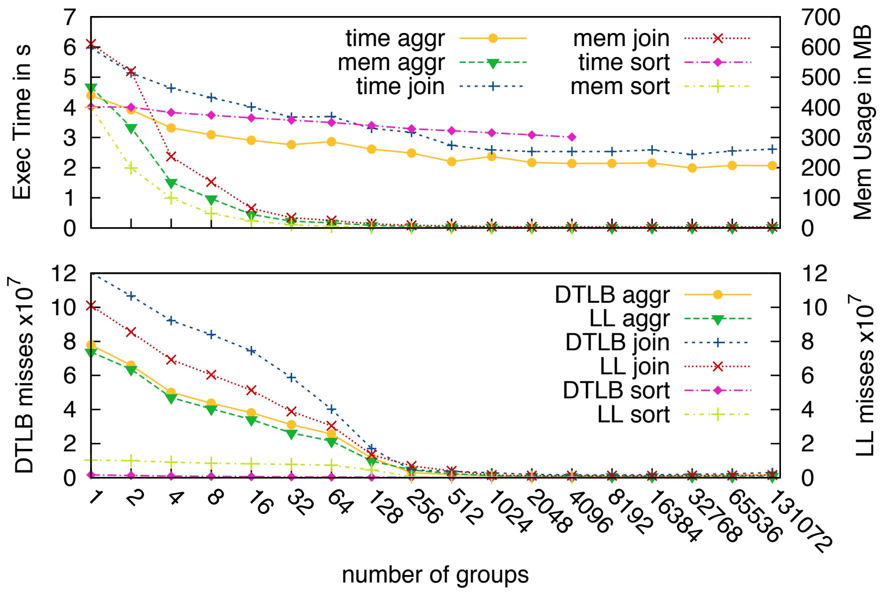


Fig. 3. Sandwiched `HashAggr` & `HashJoin`: Elapsed time, memory usage, DTLB and last level cache misses for counting the frequency of TPC-H `l_orderkey` and joining `LINEITEM` and `ORDERS` on `orderkey` using different number of groups.

of 4 Intel X25M SSDs with a stripe size of 128KB (32KB chunks per disk) and a maximum bandwidth of 1GB/s. As our implementation does not yet support parallelization, queries are only executed on a single core. Out test database system is Vectorwise. It is set up to use 4GB of buffer space and 12GB of query memory. The page size was set to 32KB. The group size of consecutively stored pages was set to 1, leaving the distribution of pages to the file system.

6.1 Micro Benchmarks

Table setup. For the micro-benchmarks for `Aggregation/Grouping` and `HashJoin` we used the `ORDERS` and `LINEITEM` table as explained in Sect. 5.3. For the `Sort` micro benchmark we used an `ORDERS` table ordered on just `o_orderdate`. Data was stored uncompressed and hot. In order to get the different number of groups, we combined two neighboring groups from one run to the next.

`Aggregation/Grouping` and `HashJoin`. The aggregation micro-benchmark scans `l_orderkey` and counts the frequency of each value. As `l_orderkey` determines the full ordering we can use the full pre-ordering for sandwiched aggregation. The join micro-benchmark performs a sandwiched hash-join of `LINEITEM` with `ORDERS` on their foreign key relationship `[l_orderkey, o_orderkey]`, with the smaller relation `ORDERS` as build relation. The time to scan the buffered relations is negligible, i.e. about 0.2s for `LINEITEM` and 0.05s for `ORDERS`. The same

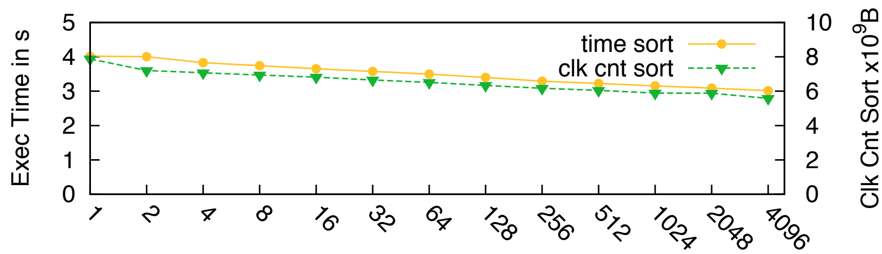


Fig. 4. Sandwiched `Sort` : Elapsed time vs. pure sorting cost for sorting `ORDERS` on `o_orderdate` using different number of groups.

holds for memory consumption, where even in case of 128k groups still 95% of memory is allocated by the aggregation or join.

Their behavior is nearly identical. The upper part of Figure 3 shows that with more groups, memory consumption goes down while speed goes up. This is explained by the lower part: hash table size decreases with higher group numbers, causing the number of TLB and lowest level cache (LL) misses to drop. At 128 groups cache misses reach a minimum, as the hash table then fits into cache (15M distinct values; $15\text{M}/128 * 32\text{B} \approx 3.6\text{MB}$).

Again, the reminder, that input relations already arrive in a cache friendly order and sandwich operators just amplify this cache residency effect. When comparing the `HashJoin` experiment to the example given in Section 5.3 where we can see the execution time and memory for 1 and 128k groups, it is obvious that a) the case with one group is faster and b) the case with 128k groups is slower. The explanation for a) are pipelining effects that accelerate the `LINEITEM-ORDERS` join in its build phase, as the tuple vectors from the `ORDERS-CUSTOMER` join are already in cache, accelerating the build phase by 60%. The explanation for b) is that more data is handled (two attributed from the customer relation and `o_orderdate`, that add memory requirement and a penalty to the execution time, becoming more visible in the cache critical experiment.

`Sort.` The `Sort` micro benchmark sorts `ORDERS` on `o_orderdate` and `o_custkey` exploiting pre-ordering on `o_orderdate`. Again, neighboring groups were combined to get different granularities. The `Sort` analysis is a bit different, as cache miss numbers are between one and two orders of magnitude lower and thus there is less impact on the execution time (see Fig. 3). Detailed profiling information, however, show that the `Sort` operator is dominated by the quick sort routines (78-82%, depending on the number of groups), and that the actual work by the CPU in these routines decreases at about the rate the execution time decreases and savings by memory access are only of minor importance (comp. Fig. 4).

6.2 TPC-H Benchmark

We implemented a prototype z-ordering [8] in Vectorwise and executed the 22 TPC-H queries scale factor SF100 one time using Sandwich Operators and an-

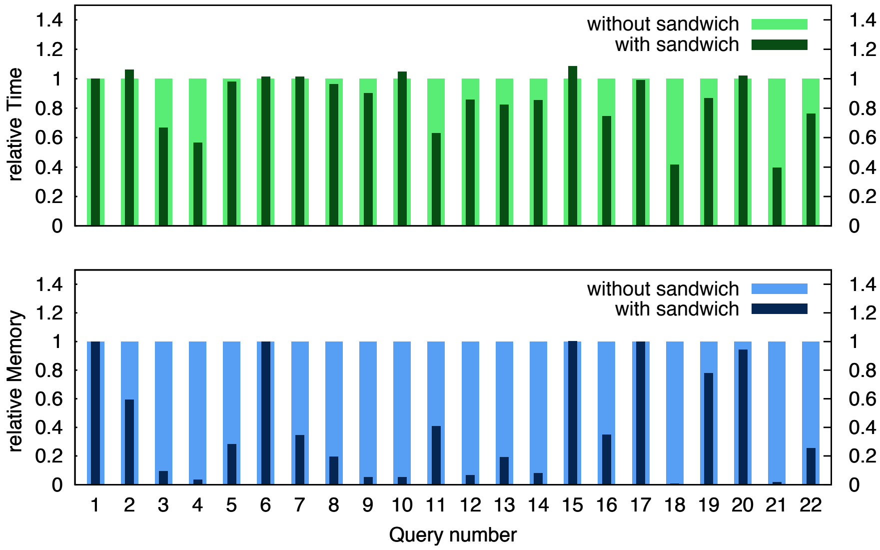


Fig. 5. Relative execution time and memory consumption for all 22 TPC-H queries with and without Sandwich Operators.

other time not using them. Besides sandwiching, all other optimizations, e.g. selection pushdown to scans, were applied in both runs. This leads to Fig. 5, where we show the differences in execution time and memory consumption for both runs. Showing clear benefits for the run with the Sandwich Operators for memory consumption as well as execution time across the query set. In total Sandwich Operators save about 125 sec and 22.4 GB of memory.

7 Related Work

In [1] special query processing techniques for MDC [11] based on block index scans are explained, that pre-process existing block-indexes before joins or aggregations and are only very briefly described. [4] techniques focus on processing partitioned data and, thus, have separate group matching and group processing phases. Both approaches differ from our approach as Sandwich Operators are fully integrated in the query plan. Systems like [15, 9] generate partition wise operators, where our approach reuses the same operator, saving memory and time. Works like [12] focus on dynamic partitioning rather than exploiting orderings in relations.

8 Conclusion and Outlook

In this paper we introduced the “Sandwich Operators”, an easy and elegant approach to exploit grouping or ordering properties of relations during query processing, that fits many table partitioning, indexing, clustering and ordering approaches, and though its treatment as grouping as a logical ordering property avoids plan size explosion as experienced in query optimization for partitioned tables. We showed how the sandwich operators accelerate `Aggregation/Grouping`, `HashJoin` and `Sort` and reduce their memory requirements.

As future work we see the combination the sandwich scheme with intra operator and intra query tree parallelization, where a `PartitionSplit` not only splits the input relation but distributes the groups for one or more operators among multiple cores, taking advantage of modern processor architectures.

Additionally, we left untouched the issue of query optimization for sandwiching in multi-dimensional storage schemes, where a query processor can generate tuple streams efficiently in many orders. Here, the question arises which orders to use, such that the query plan optimally profits from the sandwiching.

References

1. B. Bhattacharjee, S. Padmanabhan, T. Malkemus, T. Lai, L. Cranston, and M. Huras. Efficient query processing for multi-dimensionally clustered tables in db2. In *VLDB*, 2003.
2. W.-J. Chen, A. Fisher, A. Lalla, A. McLauchlan, and D. Agnew. *Database Partitioning, Table Partitioning, and MDC for DB2 9*. IBM Redbooks, 2007.
3. G. Graefe. Partitioned b-trees - a user's guide. In *BTW*, pages 668–671, 2003.
4. H. Herodotou, N. Borisov, and S. Babu. Query Optimization Techniques for Partitioned Tables. In *SIGMOD*, 2011.
5. D. Inkster, P. Boncz, and M. Zukowski. Integration of VectorWise with Ingres. *SIGMOD Record*, 40(3), 2011.
6. H. Leslie, R. Jain, D. Birdsall, and H. Yaghmai. Efficient Search of Multi-Dimensional B-Trees. In *VLDB*, 1995.
7. S. Manegold, P. Boncz, and M. Kersten. Generic Database Cost Models for Hierarchical Memory. In *VLDB*, 2002.
8. V. Markl. *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*. Institut für Informatik der TU München, 1999.
9. T. Morales. Oracle Database VLDB and Partitioning Guide, 11*g* Release 1 (11.1). Oracle, July 2007.
10. P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *TPCTC*, 2009.
11. S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras. Multi-dimensional Clustering: A New Data Layout Scheme in DB2. In *SIGMOD*, 2003.
12. N. Polyzotis. Selectivity-based Partitioning: A Divide-and-Union Paradigm for Effective Query Optimization. In *CIKM*, 2005.
13. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1976.
14. M. Stonebraker et al. C-Store: A Column-Oriented DBMS. In *VLDB*, 2005.
15. R. Talmage. Partitioned Table and Index Strategies Using SQL Server 2008. MSDN Library, March 2009.
16. X. Wang and M. Cherniack. Avoiding Sorting and Grouping in Processing Queries. In *VLDB*, 2003.
17. M. Zukowski, P. A. Boncz, N. J. Nes, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 28(2):17–22, June 2005.