University of Warsaw
Faculty of Mathematics, Computer Science and Mechanics

VU University Amsterdam
Faculty of Sciences

Joint Master of Science Programme

Michał Świtakowski

Student no. 248232 (UW), 2128012 (VU)

Integrating Cooperative Scans in a column-oriented DBMS

Master's thesis
in COMPUTER SCIENCE

First reader
Peter A. Boncz
Centrum Wiskunde & Informatica
VU University Amsterdam

Second reader
Frank J. Seinstra
Dept. of Computer Science,
VU University Amsterdam

Supervisor
Marcin Żukowski
VectorWise B.V.

August 2011

Supervisor's statement

Hereby I confirm that the present thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the present thesis was prepared by me and none of its contents was obtained by means that are against the law.

I also declare that the present thesis is a part of the Joint Master of Science Programme of the University of Warsaw and the Vrije Universiteit in Amsterdam. The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Data warehousing requires huge amounts of data to be stored on persistent storage. The usual way to access this data is to perform a sequential scan that loads and processes all the data in order it is stored. In many application areas including business intelligence, data mining and decision support systems there are multiple concurrent queries touching a considerable fraction of a relation. In most database management systems the data is accessed using a „Scan" operator that sequentially issues page request to the underlying buffer manager that caches most recently used data. In a concurrent environment, this approach may lead to suboptimal utilization of the available disk bandwidth, as the buffer manager may evict pages that could soon be reused by an another running scan.

Cooperative Scans is a new algorithm that improves performance of concurrent scans by identifying and exploiting sharing opportunities arising in a concurrent workload. The main objective of this thesis is to investigate how Cooperative Scans can be incorporated into the VectorWise database management system. Moreover, we research new areas where Cooperative Scans can be applied, such as order-aware operators. Finally, we develop the Predictive Buffer Manager, a new algorithm that uses some of the ideas of Cooperative Scans, but enables easy implementation providing similar benefits. Our experiments show significant gain in the performance achieved by the proposed solutions. However, we argue that the relative performance improvement strongly depends on the sharing opportunities arising in the workload.

Keywords

buffer management, cooperative scans, cooperative merge join, predictive buffer manager

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

H.2.4 Systems: Query Processing

Contents

Chapter 1

Introduction

1.1. Motivation and problem statement

This thesis is a result of research conducted at the VectorWise company, part of Ingres Corporation. The main area of focus is improving performance of the VectorWise Database Management System (DBMS) under I/O intensive concurrent workloads. The research was motivated by the fact that improving performance of a DBMS in this respect may lead to substantial gain in the performance perceived by the user in many applications. The development of modern computer hardware suggests an increasing importance of I/O as a bottleneck in many user scenarios. Thus, buffering of data in RAM memory and increasing the buffer reuse remains an important aspect of research in the field of Database Management Systems. To achieve our goal, we investigate incorporating a novel approach to buffer management called Cooperative Scans. Moreover, we extend Cooperative Scans to be fully integrated with the VectorWise DBMS, as well as compare it with alternative solutions.

Cooperative Scans is a framework that aims at improving the performance of concurrent scans by introducing a flexible scheduling policy that takes information about the current workload of database queries into account and allows data requested by a query to be delivered out-of-order. Scans working in parallel cooperate instead of being treated in isolation as it is done in many database management systems with a traditional buffer manager. Cooperative Scans were originally introduced in MonetDB/X100 system as described in [ZHNB07, ZBK04, Zuk09]. The motivation for researching the idea of Cooperative Scans in MonetDB/X100 was suboptimal buffer use in many application areas. The area of applicability of MonetDB/X100 includes business intelligence, decision support systems, interactive reporting, and data mining. The exhibited workload involves scanning large amounts of data in a sequential manner. Furthermore, it is common that queries are run in parallel, each performing one or more scans. In such scenarios, the commonly used buffer management policies often turn out to be sub-optimal [CD85, CR93, FNS91].

The objective of Cooperative Scans is to improve the perceived I/O bandwidth and query latency by exploiting the fact that multiple queries running in parallel can share data loaded from storage. The concept proved to be useful by attaining a considerable improvement of performance, notably in workload with many I/O bound queries. However, the implemented prototype was created only for research purposes and made many assumptions that simplified the algorithm and its interaction with the system.

The objective and novelty of this work is to investigate how Cooperative Scans can be implemented and introduced into a fully-functional DBMS taking all requirements of a modern DBMS into consideration. The challenge is to provide all functionality necessary for the

VectorWise DBMS and to not lose performance of the prototype created for MonetDB/X100. Moreover, it opens a way for interesting research related to other elements of the DBMS that can exploit characteristics of Cooperative Scans to further improve performance.

Objective. Design and implement the Cooperative Scans framework and generalize it to enable higher-level operations to benefit from it.

In this master thesis we:

(i) implement and integrate Cooperative Scans in the VectorWise DBMS

(ii) present the Predictive Buffer Manager (PBM), an alternative, simple extension to the naive buffer manager in VectorWise that turns out to provide much of the benefits of Cooperative Scans and is much less complex

(iii) introduce and implement Cooperative Merge Join as a generalization of Cooperative Scans providing support for join processing on ordered data

(iv) evaluate the performance of created optimizations on large and concurrent workloads

The thesis contains detailed description of Cooperative Scans and problems arising in the implementation and integration with the VectorWise DBMS. It presents solutions to the software-engineering problems, as well as discusses issues related to expected performance gain. The thesis also contributes an extensive evaluation of Cooperative Scans in terms of the implementation and achieved performance. It finally analyzes trade-offs related to the inclusion of Cooperative Scans in the VectorWise DBMS.

The conducted research proves both Cooperative Scans and the Predictive Buffer Manager to be a solution providing a considerable performance gains and worth including in a modern DBMS. However, the comparison with other solutions suggests that in a column-oriented DBMS a similar improvement can be achieved with less complex solutions.

The rest of the thesis is organized as follows. In Chapter 2 we present the VectorWise DBMS and stress aspects relevant for the implementation of Cooperative Scans. Chapter 3 describes the idea of Cooperative Scans. In Chapter 4 specific details concerning the implementation of Cooperative Scans are discussed. Also, Chapter 4 contains the description of the Predictive Buffer Manager that was created as a part of the implementation of Cooperative Scans. In Chapter 5 we present Cooperative Merge Join, an extension of Cooperative Scans. Chapter 6 evaluates all introduced solutions with synthetic benchmarks, whereas Chapter 7 contains the final performance evaluation with the TPC-H benchmark. Finally, Chapter 8 concludes and discusses future work.

Chapter 2

The VectorWise DBMS

In this chapter we introduce basic notions in the area of databases and Database Management Systems. Later, we present a selection of functionalities found in VectorWise that influence the design and implementation of Cooperative Scans. A more detailed description of VectorWise can be found in [Zuk09]. Moreover, we discuss what is the impact of these issues on Cooperative Scans.

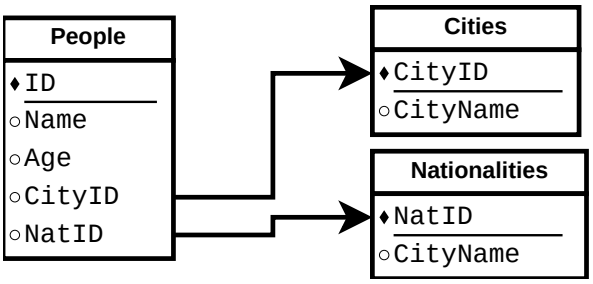
2.1. Introduction to databases and DBMS

2.1.1. Databases and data models

A *database* is an organized collection of data. Database is stored in digital form according to a certain data model. A data model reflects relevant aspects of reality and defines relationships between them. A Database Management System (DBMS) is a software package that provides a reliable way to store data in a database and process it. One of the main requirements of a DBMS is to provide a coherent interface that allows application developers to manipulate data stored in a database. For that purpose, most DBMSs use the Structured Query Language (SQL) [CB74] that allows specifying the organization of the data, inserting, deleting and querying the data.

The most wide-spread data model used by DBMSs is the relational model [Cod70]. In this model data is represented as a set of *relations*. A relation consists of a number of *attributes*. Each attribute has a specified type. The type defines what are the allowed values of the attribute e.g. an integer, a string, a date etc. A tuple is an ordered list of values of all attributes belonging to a certain relation. A set of tuples is traditionally represented as a table, where rows represent tuples and columns represent attributes.

A database schema defines how data is divided into tables and what relationships there are between the tables. An example of a database schema and a database organized according to it is depicted in Figure 2.1. The database in Figure 2.1 consists of three tables. Each of them has an attribute that serves as an unique identifier of tuple (ID, CityID and NatID in the People, Cities and Nationalities tables respectively). Moreover, the schema defines relationships between tables. The NatID and CityID attributes in the People table are references to the NatID in the Nationalities table and CityID in the Cities table respectively. We say that NatID and CityID are foreign keys referencing the Nationalities and Cities tables.



(a) A simple database schema

People				
ID	Name	Age	CityID	NatID
1	Adam	34	2	1
2	Eva	23	2	2
3	Tom	53	2	3
4	Katie	12	3	1
5	Chris	44	2	1
6	Jan	64	1	3

Cities	
CityID	CityName
1	*'Warsaw'*
2	*'Munich'*
3	*'Amsterdam'*

Nationalities	
NatID	NatName
1	*'Dutch'*
2	*'German'*
3	*'Polish'*

(b) A simple database represented as a collection of tables organized according to schema in Figure 2.1 (a)

Figure 2.1: Example of a database schema and a database organized according to it.

2.1.2. Architecture of a DBMS

The architecture of most DBMSs follows a multilayer approach, where each layer is responsible for other phase of processing. The main layers of the VectorWise DBMS include:

(i) client application – which issues a query to the DBMS

(ii) query parser – which parses the text representation of a query and creates an internal representation used by the DBMS

(iii) query optimizer – which tries to find an efficient way to execute the query and creates a query plan

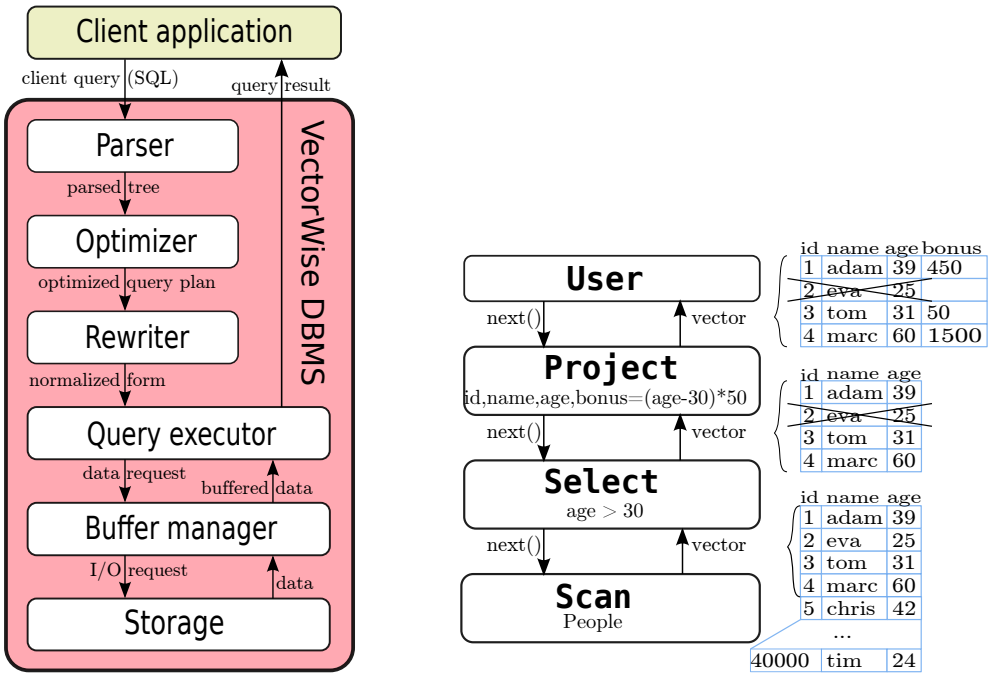
(iv) query rewriter – which improves the query plan by applying chosen transformations

(v) query executor – which does the actual processing of the data

(vi) buffer manager – that loads data from the storage and buffers it in the memory for processing

(vii) storage – which handles storing data on persistent storage device such as hard disk

Figure 2.2 (a) depicts the above-mentioned layers of the systems and shows interactions between them. In this master thesis we mostly focus on the buffer manager layer. Cooperative Scans influence the way data is loaded into the buffer and the decisions which data to keep



(a) Multilayer architecture of the VectorWise DBMS

(b) Processing of a query in Volcano model

Figure 2.2: Organization of the VectorWise DBMS

in memory. Moreover, we had to extend the Rewriter layer with transformations that need to be applied to the query plan to actually use the Cooperative Scans framework.

2.2. VectorWise DBMS

The VectorWise DBMS (www.vectorwise.com) is an analytical database management system. It is mainly used for applications such as online analytical processing, business intelligence, decision support systems and data mining. The VectorWise DBMS distinguishes itself from other systems by exploiting features and characteristics of modern hardware such as SIMD instructions, out-of-order execution, on-chip cache memory and multiple cores per one socket. To benefit from those features VectorWise uses a novel vectorized in-cache execution model.

2.3. Vectorized execution model

The query execution engine of the VectorWise DBMS uses the typical Volcano-like „iterator" model [Gra94] used in many other DBMS architectures. A query is represented as a tree of operators. We refer to it by the Query Execution Tree (QET). Each operator can be seen as an object performing a part of processing of the data. Every operator has three methods: `open()`, `next()` and `close()`. The `open()` method starts a new stream of data, `next()` fetches a new tuple and finally `close()` finishes the processing. Contrary to traditional database management systems, VectorWise uses a novel „vectorised" model [BZN05]. The `next()` method returns a vector of tuples instead of a single tuple. A vector is a small in-memory array of values, with a typical size of 100-1000 elements. Thus, the operators in VectorWise can process multiple tuples at a time using long loops iterating over a vector of

values. This approach has several advantages in terms of performance. First of all, processing in long loops results in amortization of overhead related to function calls. Secondly, better CPU cache utilization is achieved provided the processed vectors fit the CPU cache. Lastly, processing all values in a tight loop enables the compiler to exploit SIMD instructions that can be found in modern processors. As a result, VectorWise achieves significantly lower instruction per tuple ratio, which has a strong impact on the overall performance of the system.

2.4. Column Storage

VectorWise uses a flexible form of columnar storage. Data is stored in pages of the same, fixed size. Every page consists of consecutive values belonging to one column of a certain table. The column-based approach is often referred to as the Decomposition Storage Model (DSM) [CK85b], whereas row-based storage as N-ary Storage Model (NSM). Another approach that tries to combine advantages of NSM and DSM is PAX [ADHS01]. It is a generalization of DSM allowing multiple columns to be stored in one page. Inside the page data of each column is stored consecutively as in DSM. VectorWise uses the PAX format for storing small tables that can fit in a single page. Thus, in this thesis focus on DSM storage, as Cooperative Scans are aimed at improving processing of vast amounts of data.

DSM has several advantages over the traditional row storage. First of all, DSM uses less I/O bandwidth as a query scans only a subset of columns that is necessary for query processing. Consequently, VectorWise avoids overhead of extracting unneeded attributes, which arises in row-based DBMSs. Secondly, storing consecutive values of the same type enables efficient, light-weight compression techniques such as RLE, PDICT, PFOR, PFOR-DELTA, which results in further I/O savings [ZHNB06, Zuk09].

On the other hand, column storage is less efficient when it comes to updating the database. For instance, adding a single tuple requires multiple disk writes – one for each column of the table. To solve this problem VectorWise uses special in-memory data structures called Positional Delta Trees [HZN+10] (see Section 2.8).

DSM storage poses several challenges to the implementation of Cooperative Scans. The most fundamental issue is that the data volume shared by concurrently working queries is reduced in DSM storage [ZHNB07]. Two queries can share I/O bandwidth only if they scan overlapping sets of columns. Figure 2.3 presents a comparison of the number of disk pages accessed by both of the two example queries in NSM and DSM approach. As we can see the number of pages shared by the two queries is lower in DSM, as only pages storing data of columns c and f can be shared.

Another issue is the fact that in DSM the number of values stored in a single page varies depending on the column type. For instance, a page with strings can contain several thousands of values whereas a page with integers is able to hold up to several millions of values. As a consequence, it is virtually impossible to create horizontal partitioning of a table that would be aligned to page boundaries. Loading a range of tuples in DSM is first translated to loading a range of pages, which is different for each column. So, for each column a superset of the requested range will be loaded.

2.5. Min-max indices

Besides DSM storage and compression, VectorWise uses min-max indices to decrease the need for I/O operations. In short, a min-max index is an equi-width histogram of a certain attribute.

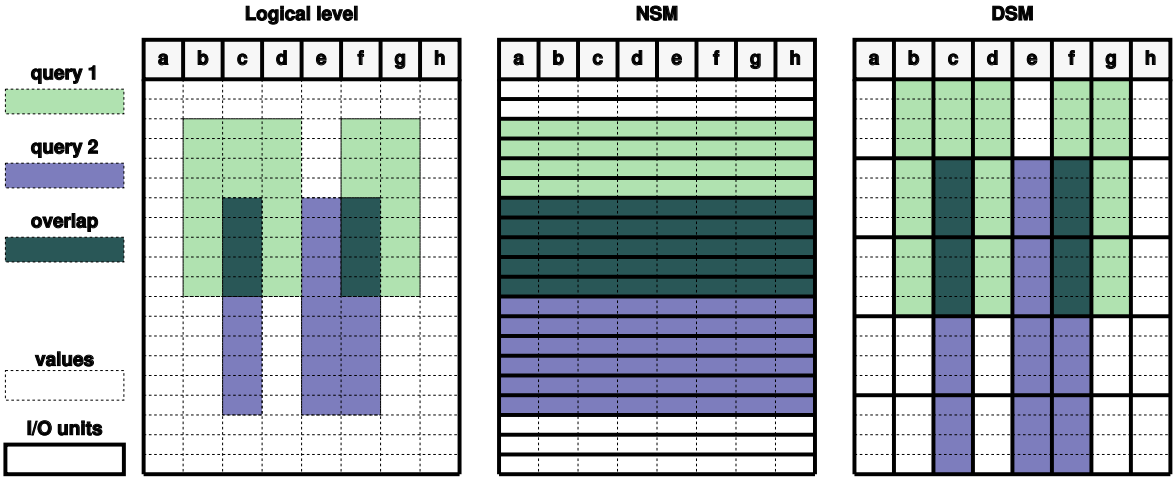


Figure 2.3: Overlap of data requested by two example queries in NSM and DSM (from [Zuk09]).

A column is divided into up to M groups, where M is a constant defined at the time of creation of the database. Thus, for large databases a group may contain several millions of tuples needing several MB of disk space to be stored. For each group we store boundary tuple identifiers along with minimal and maximal value of the indexed attribute. This metadata is stored in a summary table that is small enough to be kept in RAM memory at all times.

Figure 2.4 shows a part of the lineitem table, which is a part of the TPC-H benchmark [Tra11] along with min-max indices for two of its columns: shipdate and linenr.

To explain the principles of min-max indices we consider the following SQL queries:

```
SELECT * FROM lineitem
    WHERE shipdate BETWEEN '1997-11-1' AND '1998-1-17';
```

By looking at the min-max index on the shipdate column we can derive that the range of tuples to be loaded can be trimmed from [0, 21) (the full range) to [0, 10), thus saving half of the originally needed volume of I/O.

```
SELECT * FROM lineitem
    WHERE linenr BETWEEN 1 AND 6;
```

In this case the min-max index on the linenr column does not allow to optimize the amount of data to be loaded from disk. As we can see, all buckets in the min-max index contain tuples failing into the requested range of values of the linenr column.

To sum up, the amount of I/O that can be avoided thanks to min-max indices depends on the ordering of the table. The order does not need to be perfect. As long as there is correlation with the tuple order, min-max indices are effective. In the presented example the table is almost sorted on the shipdate column, whereas the distribution of values in the linenr columns is less predictable. As a result, min-max indices perform well in queries with selection on shipdate and can fail to do so in queries with selection on linenr.

2.6. Join indices and cluster trees

Join indices are not a new concept in relational databases, they were first introduced in [Val87]. An abstract definition of a join index is as follows: We consider two relations A and B. Every tuple in a relation has a unique identifier that is denoted as a_i and b_i respectively.

Min-max index on lineitem.shipdate			
Bucket ID	ID	Min value	Max value
0	0	1997-12-1	1997-12-22
1	5	1997-12-26	1998-1-20
2	10	1998-1-21	1998-4-23
3	15	1998-4-7	1998-7-13

Min-max index on lineitem.linenr			
Bucket ID	ID	Min value	Max value
0	0	1	4
1	5	2	5
2	10	6	11
3	15	1	14

lineitem				
ID	orderkey	shipdate	receiptdate	linenr
0	order001	1997-12-1	1997-12-3	1
1	order001	1997-12-18	1997-12-30	2
2	order001	1997-12-20	1997-12-25	3
3	order002	1997-12-21	1998-1-10	1
4	order001	1997-12-22	1998-1-2	4
5	order002	1997-12-26	1998-1-5	5
6	order002	1998-1-15	1998-2-7	2
7	order002	1998-1-16	1998-2-1	3
8	order002	1998-1-20	1998-3-13	4
9	order002	1998-1-20	1998-3-5	5
10	order002	1998-1-21	1998-4-23	6
11	order003	1998-1-23	1998-2-1	7
12	order003	1998-4-13	1998-4-12	9
13	order004	1998-4-20	1998-6-10	10
14	order004	1998-4-23	1998-5-2	11
15	order003	1998-4-7	1998-4-20	8
16	order004	1998-5-23	1998-6-9	12
17	order004	1998-5-24	1998-6-2	13
18	order004	1998-6-3	1998-6-10	14
19	order004	1998-6-2	1998-6-20	1
20	order004	1998-7-13	1998-7-20	2

Figure 2.4: The lineitem table with min-max indices

We define a join index as a set:

$$JI = \{(a_i, b_i) | f(a_i, b_i)\ is\ true\}$$

As we can see, a join index is an abstraction of the join of two relations. Figure 2.5 shows the lineitem and orders tables from the TPC-H benchmark [Tra11] and a join index between them. The ID column in both tables acts as unique identifier. The join index is represented as table with two columns: srcPos and dstPos. The srcPos column contains references to the ID column of the orders table, whereas dstPos references ID column of the lineitem table.

Usually database management systems exploit join indices by using a nested loop index join. This algorithm uses $log(N)$ index search to find matching tuples, which results in random disk accesses. This is unacceptable in VectorWise as it uses DSM storage with large page size, thus finding a single tuple would require many I/O requests and loading a lot of not needed data.

To solve this problem the ordering should be exploited to be able to use the merge-join algorithm. The merge-join algorithm processes two data streams ordered on a specified key. Merge-join finds matching tuples by performing two interleaved scans on both streams. The first or the second scan is advanced to find the next match. In particular, if both inputs are sorted in ascending order, the next match is found by advancing the input with lower current value. Appropriate ordering on the join equi-key of both input streams can be achieved by sorting. This however, is almost always slower than alternative hash-join algorithm. The only viable scenario where sorting combined with merge-join can outperform hash-join is a join between a small unsorted stream and a large sorted stream. For the details of hash-join algorithm please refer to Section 2.10.5.

However, in practice we encounter joins between two or more large tables. In particular in the TPC-H benchmark we often have to join the two largest tables, namely lineitem and

orders		
ID	orderkey	orderdate
0	order001	1997-11-10
1	order002	1997-12-23
2	order003	1998-6-2
3	order004	1998-8-23

Join index	
srcPos	dstPos
0	0
0	1
0	2
0	4
1	3
1	5
1	6
1	7
1	8
1	9
1	10
2	11
2	12
2	15
3	13
3	14
3	16
3	17
3	18
3	19
3	20

lineitem				
ID	orderkey	shipdate	receiptdate	linenr
0	order001	1997-12-1	1997-12-3	1
1	order001	1997-12-18	1997-12-30	2
2	order001	1997-12-20	1997-12-25	3
3	order002	1997-12-21	1998-1-10	1
4	order001	1997-12-22	1998-1-2	4
5	order002	1997-12-26	1998-1-5	5
6	order002	1998-1-15	1998-2-7	2
7	order002	1998-1-16	1998-2-1	3
8	order002	1998-1-20	1998-3-13	4
9	order002	1998-1-20	1998-3-5	5
10	order002	1998-1-21	1998-4-23	6
11	order003	1998-1-23	1998-2-1	7
12	order003	1998-4-13	1998-4-12	9
13	order004	1998-4-20	1998-6-10	10
14	order004	1998-4-23	1998-5-2	11
15	order003	1998-4-7	1998-4-20	8
16	order004	1998-5-23	1998-6-9	12
17	order004	1998-5-24	1998-6-2	13
18	order004	1998-6-3	1998-6-10	14
19	order004	1998-6-2	1998-6-20	1
20	order004	1998-7-13	1998-7-20	2

Figure 2.5: The lineitem and orders tables along with join index

orders, which are joined on the orderkey attribute. As discussed in Section 2.10.5, the hash-join algorithm may consume considerable amount of memory to store the inner relation in a hash table.

To solve the above-mentioned problems, VectorWise uses clustering. The idea is to store one table in physical order corresponding to the tuple order in the other table which is referenced by the first table with a foreign key. We say that the referenced table is a parent table and the referencing table is a child table. In Figure 2.6 we can see the orders table ordered on orderkey and the lineitem table clustered on orderkey along with the clustered version of a join index and min-max indices on lineitem as discussed below.

Clustering allows to store the join index in a simpler way. There is no longer a need to store a separate table with two columns. Instead, we can add an additional column to either parent or child table. Both solutions are depicted in Figure 2.6. The first solution is to store $join\_index.srcPos$ as an additional column of the child table (column JI in the lineitem table in Figure 2.6). The other possibility is to store the number of matching tuples in child table for each tuple on parent side (column JI in the orders table in Figure 2.6). This is in fact a RLE-compressed form of the $join\_index.srcPos$ column. VectorWise employs the second variant. However, both versions are available to the query processing layer – the uncompressed form can be created on-the-fly while scanning the child table.

2.6.1. Cluster trees

The idea of clustering two tables may be easily extended to multiple tables. We can treat a table as a node in a tree. Nodes are connected if and only if two tables are in parent-child relationship. That means they are connected by means of a foreign key and clustered on that key. Figure 7.2 depicts an example cluster tree build based on the TPC-H [Tra11] schema (see Section 7.1.1).

orders			
ID	orderkey	orderdate	JI
0	order001	1997-11-10	4
1	order002	1997-12-23	7
2	order003	1998-6-2	3
3	order004	1998-8-23	7

Min-max index on lineitem.shipdate			
Bucket ID	ID	Min value	Max value
0	0	1997-12-1	1997-12-26
1	5	1997-12-21	1998-1-20
2	10	1998-1-21	1998-4-20
3	15	1998-4-23	1998-7-13

Min-max index on lineitem.linenr			
Bucket ID	ID	Min value	Max value
0	0	1	5
1	5	1	5
2	10	6	10
3	15	1	14

lineitem					
JI	ID	orderkey	shipdate	receiptdate	linenr
0	0	order001	1997-12-1	1997-12-3	1
0	1	order001	1997-12-18	1997-12-30	2
0	2	order001	1997-12-20	1997-12-25	3
0	3	order001	1997-12-22	1998-1-2	4
1	4	order002	1997-12-26	1998-1-5	5
1	5	order002	1997-12-21	1998-1-10	1
1	6	order002	1998-1-15	1998-2-7	2
1	7	order002	1998-1-16	1998-2-1	3
1	8	order002	1998-1-20	1998-3-13	4
1	9	order002	1998-1-20	1998-3-5	5
1	10	order002	1998-1-21	1998-4-23	6
2	11	order003	1998-1-23	1998-2-1	7
2	12	order003	1998-4-7	1998-4-20	8
2	13	order003	1998-4-13	1998-4-12	9
3	14	order004	1998-4-20	1998-6-10	10
3	15	order004	1998-4-23	1998-5-2	11
3	16	order004	1998-5-23	1998-6-9	12
3	17	order004	1998-5-24	1998-6-2	13
3	18	order004	1998-6-3	1998-6-10	14
3	19	order004	1998-6-2	1998-6-20	1
3	20	order004	1998-7-13	1998-7-20	2

Figure 2.6: The lineitem and orders tables clustered

2.6.2. Join Index Summary

There are situations when we need to find specified positions in the join index. Most frequently, we need only a part of the joined result. To allow efficient retrieval of the partial join of two tables, VectorWise implements a sparse index on the join index called the Join Index Summary (JIS). Similarly to min-max indices the JIS contains at most M entries, where M is a configurable parameter. Let us assume that a sparse index on the uncompressed join index is created, i.e. the join index is divided into buckets. Each entry in the JIS contains two values: parent table ID and child table ID. To conserve space, only the lower boundaries are stored, the upper boundary is simply the lower boundary of the next bucket. Thus, the last entry in the JIS is always equal to the total number of tuples in the parent and the child table.

In VectorWise, the sparse index over the join index has an extra property. It is clustered on the join key. That means that tuples with the same value of foreign key (FK) always fall into the same bucket. This property turns out to be very important for the correct implementation of Cooperative Merge Join (see Section 5.2).

An example JIS that is build based on the join index in Figure 2.6 is depicted in Figure 2.7.

2.6.3. Exploiting min-max indices, JIS and clustering

In many cases, min-max indices allow to minimize the volume of I/O that needs to be performed for a certain query. This feature can also be exploited in a scenario where join indices are used. Let us consider the following SQL query:

```
SELECT * FROM lineitem, orders
    WHERE lineitem.orderkey = orders.orderkey
    AND BETWEEN '1997-12-18' AND '1998-1-15';
```

Join index summary	
Parent SID	Chilld SID
0	0
2	11
3	14

Figure 2.7: Join index summary for the lineitem and orders tables

Thanks to the min-max index on the shipdate column we can derive that only tuples from the lineitem table with ID in range [0, 10) need to be loaded. Furthermore, we can use the JIS to translate this range from the child table (lineitem) to the parent table (orders). Indeed, tuples in range [0, 10) fall into the first bucket of the JIS depicted in Figure 2.7. Thus, on the parent side we only need to consider tuples in range [0, 2), namely only the first two tuples. As we can see, combining min-max indices, clustering and the JIS can significantly reduce the need for I/O.

2.7. Snapshot isolation

Transactional functionality of database modifications is one of the most fundamental features of modern database management systems. A transaction is defined as a unit of work performed on a database in a coherent and reliable way. A transaction must be atomic, consistent, isolated and durable (ACID). We focus on isolation of transactions which is the most relevant aspect for Cooperative Scans.

The VectorWise DBMS adopts snapshot isolation [BBG$^+$95] to guarantee that every transaction operates on a consistent state of the database. Snapshot isolation is a mechanism that guarantees that each transaction sees a consistent view of the database. The start of a transaction involves creating a snapshot that consists of all changes made by the last committed transactions. A transaction working on its own snapshot does not influence working of concurrent transactions. In particular, all modifications performed by a transaction are done to its own snapshot. Before a transaction commits, a validation step is performed to check whether updates conflict with any other commit made by a concurrent transaction since transaction's snapshot was created. A transaction is either fully committed or aborted depending on the result of the validation. The committed changes form the master snapshot. Each new transaction that starts its execution creates its own copy of the master snapshot that it may subsequently modify.

The aim of Cooperative Scans is to exploit sharing opportunities that arise with concurrently working queries. Thus, in spite of the existence of the local snapshots of each transaction that may differ, Cooperative Scans should still be able to correctly identify sharing opportunities.

2.8. Positional Delta Trees

VectorWise uses DSM storage, which is best suited for read-only workloads. To solve the problem of costly updates in DSM, a novel in-memory data structure called Positional Delta Tree (PDT) has been introduced. A PDT is a differential data structure that keeps the changes that have been made to a table. By keeping changes in PDTs, the underlying DSM tables do not have to be modified. The PDT provides logarithmic update performance with respect to the size of the PDT. Reconstructing the latest table image in queries is also efficient, thanks

to the fact that it organizes changes by position (the position in the table where the change applies). All queries must apply a PDT merging operation, so that they process up-to-date image of the table. PDTs are differential data structures, and these can be stacked denoting differences on top of differences. PDTs can form a hierarchy with a global, large PDT at the bottom and small transaction-local PDTs at the top. A detailed description of PDTs can be found in [HZN+10].

The PDT merge operation works as follows. During the scan operation, data fetched from the stable storage is merged with PDT-resident tuples from all layers i.e. new tuples are inserted into the incoming stream and deleted ones are skipped. As we can see, there is a difference between tuples stored on the disk and tuples visible to the query. Therefore it is important for Cooperative Scans to load stable data in a way that ensures correctness of the merging process.

Also, tuples stored in PDTs influence functioning of join indices, merge-join and Cooperative Merge Join in particular. We discuss join indices in detail in Section 2.6.

2.9. Checkpointing

Checkpointing is an aspect closely related to PDT updates. When updates are applied to the database, the memory usage of PDTs may grow considerably. This process can not be sustained forever. At some point, the whole table has to be rebuilt (checkpointed). This process involves scanning the table from the stable storage, merging updates stored in PDTs and storing the result as a new version of the same table. After checkpointing finishes, all incoming queries will work on the new version of the table that has just been saved. Note that for the new version of the table, a new set of pages is allocated i.e. the old and the new version do not share any pages.

As a result, checkpointing can lead to scenario where there are queries working on two versions of the same table that are completely different in terms of the set of pages they are stored in. Thus, it is far more complicated for two queries working on the old version and the new version to share loaded data.

The checkpointing process is more complicated when there is a join index on a given table. In that situation, checkpointing one table may lead to checkpointing all tables that are connected by means of a join index.

2.10. VectorWise Algebra

The VectorWise query algebra is used for internal representation of queries inside the system kernel. It is used to directly define the QET. The VectorWise algebra is based on relational algebra, a concept used in many relational database management systems. From the user's perspective, the VectorWise algebra is invisible. The user issues queries that are expressed in the SQL language which are subsequently optimized by the optimizer and translated into the VectorWise algebra. The VectorWise kernel applies further modifications to the QET by rewriting the tree into another equivalent representation (see Section 2.11).

In this section, we briefly describe operators implemented in VectorWise that influence the design of the Cooperative Scans framework. For every operator we introduce notation that will be used in further sections of this thesis.

2.10.1. Scan

The task of the Scan operator is the retrieval of data belonging to a certain table. The Scan operator creates a data stream that is subsequently processed by other operators. The data stored in a table can be trimmed both vertically and horizontally. The table can be trimmed vertically by scanning only a subset of its columns. The horizontal trimming refers to filtering out tuples whose identifiers do not belong to specified ranges. Such tuple ranges are derived using min-max indices and join index summaries as discussed in Sections 2.5 and 2.6.3.

We denote Scan by $Scan_{table\_name,column\_list,range\_list}$ where $table\_name$ is the name of a table to be scanned and $column\_list$ is a list of columns of that table. The $range\_list$ is a list of continuous ranges of tuple identifiers that will be included in the output data stream. If the $range\_list$ is not specified the whole table is scanned.

2.10.2. Select

The select operator is used to filter out tuples that do not fulfill a specified predicate.

We denote Select by $Select_{bool\_expr}$ where $bool\_expr$ is an arbitrary boolean expression used to choose matching tuples.

2.10.3. Aggregation

Aggregation allows to group tuples with the same value of a certain attribute and perform calculations on them. Calculations that can be performed in VectorWise include sum, min, max and count. The average (avg) is simulated by a combination of sum and count.

A typical implementation of aggregation involves a hash table which is used to group tuples with the same key together. It should be noted that this approach requires materialization of the output. Materialization refers to processing schema where all tuples are processed before being passed to the parent operator. This approach does not scale well in terms of memory consumption. Therefore, materialization may become a problem when the number of distinct aggregate key values is large.

We denote Aggregation by $Aggr_{keys,aggr\_list}$ where $keys$ is a list of attributes that form aggregate key and $aggr\_list$ is a list of aggregate functions (sum, min, max or count).

2.10.4. Ordered Aggregation

Ordered Aggregation is an optimized version of Aggregation, which exploits the fact that data in the input stream can be clustered on a key. We say that data stream is clustered when two occurrences of the same key value are never separated by a different value. Thanks to that property, the OrderedAggregation operator can immediately return the result to the parent operator when it detects a new group value in the stream. It should be noted that ordered aggregation is considerably faster than hash-based aggregation and has lower memory consumption. Consequently, ordered aggregation is used in query plans wherever it is possible i.e. input stream is clustered.

Similarly to Aggregation we denote OrderedAggregation by $OrdAggr_{keys,aggr\_list}$ with the same meaning of $keys$ and $aggr\_list$.

2.10.5. HashJoin

HashJoin is one of the most widely used algorithms for performing the join operation in database management systems. The idea is to first build a hash table from the inner (right)

data stream. Once the hash table is built, the outer (left) data stream is processed. For each tuple, a hash value is used to check whether or not there is a matching tuple in the hash table.

It should be stressed that the hash table may consume a considerable amount of RAM memory if the inner (right) relation is large. We say that HashJoin materializes the right data stream in memory. Therefore, VectorWise also incorporates another join algorithm discussed in the next section.

An important aspect of HashJoin is the fact that it returns tuples in the same order as its left input. This is exploited in Section 4.9 and 5.5.

We denote HashJoin by $HashJoin_{keys_A,keys_B}$ where $keys_A$ and $keys_B$ are equally sized sets of columns that define the join equality predicate.

2.10.6. MergeJoin

MergeJoin is the second join operator implemented in VectorWise. MergeJoin assumes that both input streams are ordered by join keys. MergeJoin processes data in a pipelined fashion. Lack of materialization results in low memory usage. Also, MergeJoin is usually faster than HashJoin. Thus, the query optimizer in VectorWise tries to use MergeJoin in QET wherever possible. The ability to use MergeJoin in VectorWise depends on the schema. We can assume tables to be properly ordered only if tables joined over a foreign key are in the parent–child relationship in the same cluster tree. If there is no appropriate join index, VectorWise always falls back to HashJoin.

Similarly to HashJoin we denote MergeJoin by $MergeJoin_{keys_A,keys_B}$ with the same meaning of $keys_A$ and $keys_B$.

2.10.7. Sort

The Sort operator is used to sort the incoming data stream according to some order. The sort order is defined by a list of attributes, where the order of attributes corresponds to the order of sort keys. It should be noted that in VectorWise the Sort operator is mainly used to sort the output presented to the user. The query execution layer relies on data already sorted on disk as described in Section 2.6.

We denote the Sort operator by $Sort_{sort\_columns}$.

2.10.8. TopN

The TopN operator returns first N tuples according to specified sort order. Note that to compute a TopN, one does not need to sort, rather a single pass inserting tuples into a small heap suffices.

We denote TopN by $TopN_{sort\_columns,N}$.

2.10.9. Reuse

The purpose of the Reuse operator is to optimize execution of plans that contain multiple copies of the same subtree. The repeating subtree can be executed only once and the result can be buffered for future use in the query plan. Effectively, the Reuse operator allows to support directed acyclic graphs (DAGs) as query plans. In such case the root of the reused subtree has multiple parent nodes.

2.10.10. Append, Insert and Delete

VectorWise uses three basic operators for applying modifications to a database: Append, Insert and Delete. There are two different operators for adding tuples to a table. The Insert operator inserts tuples into the memory-resident Positional Delta Trees (PDTs) structures (see Section 2.8). On contrary, the Append operator adds new tuples directly to the stable storage. The Append adds new tuples to the last page of each column of the table, or adds new pages to the end of the column if the last page is full. Thus, it is only able to add tuples at the end of the table. Finally, the Delete operator deletes tuples from the table, by marking tuples as deleted in PDTs. Deleted tuples are skipped in the PDT merging process described in Section 2.8.

2.10.11. Sandwich operators

The VectorWise DBMS extends the standard `open()`, `next()`, `close()` interface of operators with an additional `restart()` method. Its purpose is to reinitialize the operator to prepare it to accept further data flow. The underlying assumption is that the flow is partitioned and different partitions exhibit different characteristics. Therefore, the operator that accepts a partitioned flow may need to reinitialize its state to be able to accept the next partition. The exact semantics of the `restart()` varies depending on the operator e.g. in hash operators hash tables are flushed. In all the sandwich operators allow to use all existing algebra operators unmodified to support partitioned stream semantics.

The new method is used by the PartitionSplit and PartionRestart operators which together form a „sandwich" operator. The idea is to enclose existing operators between PartitionSplit and PartionRestart. We say that an operator is „sandwiched" between PartitionSplit and PartionRestart. The sandwiched operator is required to implement the `restart()` method.

Partition Split

PartitionSplit divides the input stream into groups according to a certain key. Similarly to Aggregation, the key is defined as a list of attributes. PartitionSplit assumes that the input stream is clustered on the key. Thus, PartitionSplit works in a similar way to OrderedAggregation, but it does not calculate any aggregate function. Instead, it signals end of stream to its parent operator when the end of a group is detected. However, despite signaling end-of-stream to its parent there may still be new tuples coming. So, at the end-of-stream the PartitionSplit merely expects a new partition key value to appear and continues to produce more tuples when asked to do so (with the `next()` method).

We denote ParitionSplit as $PartitionSplit_{keys}$ where *keys* is a list of attributes that define the key.

Partition Restart

The PartitionRestart operator works on top of an operator that implements the `restart()` method. The underlying operator can be either unary or binary. We focus on the binary case which is more general. The task of PartionRestart is to synchronize the streams coming from underlying PartitionSplit operators. When the end of a group is detected i.e. the son of PartitionRestart signals end-of-stream (but in fact that is just an end-of-partition), Partion-Restart calls the `restart()` method of its son and corresponding PartitionSplit operators. PartitionRestart effectively performs a merge-join in groups, so that its son performs opera-

tions only matching groups. PartionRestart can be also configured to allow empty groups on one or both of the sides in a similar way outer joins are performed.

We denote ParitionRestart as $PartitionRestart_{outer\_spec}$ where $outer\_spec$ is one of $\{INNER, RIGHT\_OUTER, LEFT\_OUTER, FULL\_OUTER\}$ defining outer join logic.

2.11. Rewriter

The VectorWise employs an internal representation of query plans using the VectorWise algebra introduced in Section 2.10. Those query plans are processed by the Rewriter module. The task of the Rewriter is applying modifications to the query plan that optimize it and prepare it to be executed. Rewriting involves two main operations. First, nodes forming the query plan are annotated with properties. Properties are $(key, value)$ pairs, where key is a name of certain property and $value$ is a parameter of arbitrary type. Secondly, the Rewriter transforms the query plan according to a specified set of transformations that are triggered when certain properties are set. It can replace a single node with another one, as well as transform whole subtrees and change their shape. Tasks performed by the Rewriter include eliminating unnecessary operators, trimming ranges basing on the min-max indices (see Section 2.5) and parallelizing the query plan. The Rewriter serves as a tool that introduces various optimizations implemented in VectorWise. It is also used to introduce Cooperative Scans as described in Section 4.9 and 5.5.

2.12. Summary

This chapter presented the fundamentals of VectorWise, as well as stressed aspects relevant for the design and implementation of Cooperative Scans. It defines issues and challenges that will be addressed in the following chapters. Below, we provide a summary of those problems along with references to sections describing and addressing them.

(i) implementing Cooperative Scans on top of a columnar storage (discussed in Section 2.4, addressed in Chapter 3)

(ii) providing support for updates stored in PDT structures (discussed in Section 2.8, addressed in Section 4.5)

(iii) handling of transactions working on different snapshots (discussed in Section 2.7, addressed in Section 4.6, 4.7 and 5.6)

(iv) providing solutions for operators requiring ordered input stream (discussed in Section 2.10, addressed in Chapter 4 and 5)

(v) transforming query plans to use the Cooperative Scans framework (discussed in Section 2.11, addressed in Section 4.9 and 5.5)

Chapter 3

Cooperative Scans

This chapter describes the Cooperative Scans algorithm in detail. First, the inefficiencies of traditional approach to scan processing are discussed. It is followed by an introduction to the principles of Cooperative Scans and details of the implementation. We introduce two new concepts that will be used throughout the thesis: a new operator called CScan and the Active Buffer Manager (ABM).

3.1. Traditional scan processing

Database systems are designed to support a multi-user environment with many concurrently working queries. In such scenarios, the I/O subsystem has to deal with many requests that are issued in semi-random access patterns leading to frequent disk-arm movements. With current characteristics of hard disks, this behavior may lead to performance degradation as the I/O bandwidth drops with access patterns that require many disk-arm movements. Most DBMSs avoid this problem by executing large isolated I/O requests. As a consequence, the time of disk arm movements is well amortized and the perceived I/O bandwidth is close to a fully sequential scan. While helping with attaining good I/O bandwidth, this does not solve the problem of making the right decisions about loading, assigning and evicting data, to provide high performance under concurrent workloads.

To illustrate the need for a flexible scan scheduling policy, we consider the following example. Assume the database is composed of 30 pages numbered from 1 to 30. The system is equipped with a buffer pool that can hold up to 10 pages. To simplify calculations, we assume that processing a page that is already loaded into the memory is instantaneous, whereas loading a single page takes one second. Also, the time of loading multiple pages does not depend on the access pattern i.e. we assume the overhead of disk-arm movements to be well amortized. We consider two queries Q_1 and Q_2 reading pages from 1 to 20 and from 21 to 30 respectively. Queries Q_1 and Q_2 start at the same time. The buffer pool contains pages {16, 17, 18, 19, 20, 26, 27, 28, 29, 30} thus the last 5 pages for Q_1 and Q_2. We can consider various ways of executing those two queries with respect to scheduling of I/O requests and buffer management policy. Firstly, we can assume that the buffer is governed by a LRU policy (see Section 4.2) where page 16 is at the head of the queue and page 30 at the tail. We also assume that queries read pages sequentially in ascending order.

The three most simple rules for scheduling I/O operations are: round-robin, scheduling all Q_1's pages first and scheduling all Q_2's pages first. In the case of a round-robin policy with granularity of a single page, all pages in the buffer pool will be evicted after 10 seconds when Q_1 processes pages from 1 to 5 and Q_2 pages from 21 to 25. Thus, no data in the buffer pool

Query scheduling policy	Buffer management policy	Q_1 latency	Q_2 latency	Average latency
round-robin	LRU	40	20	30
Q_1 first	LRU	20	30	25
Q_2 first	LRU	25	5	15
out-of-order	any	20	5	12.5

Table 3.1: Comparison of different approaches to processing two simple queries

will be reused. In this scenario Q_2 will finish after 20 seconds and Q_1 after 40 seconds. If we decide to satisfy the whole Q_1 before processing Q_2, the whole buffer pool will be flushed resulting in no data reuse. In this case, Q_1 finishes after 20 seconds and $Q2$ after 30 seconds. In yet another scenario when Q_2 is serviced first, it loads pages from 21 to 25 resulting in eviction of pages needed by Q_1 (from 16 to 20). Then Q_2 can immediately process pages already loaded into the memory, but Q_1 will again have to load all needed pages. As a result, Q_2 finishes after 5 seconds and Q_1 after 25 seconds.

If we drop the implicit assumption that pages are processed in tuple order, we can further improve performance. The best solution in our example would be to let them first process all pages already kept in the buffer pool, and then let Q_2 finish by loading all the remaining pages from the disk. Finally, Q_1 should also load all its pages and finish. In that scenario, Q_2 is ready after 5 seconds and Q_1 after 20 seconds. It should be noted that the exact buffer management policy in this case is not important as all needed pages will have to be loaded and it does not matter which ones will be evicted to make space for new ones. However, it certainly influences any further queries that may be executed in the system. Table 3.1 presents a comparison of all above-mentioned approaches and their performance.

3.2. Introduction to Cooperative Scans

In this section, we present the design of the Cooperative Scans framework that aims at removing inefficiencies of the traditional scan processing. The two main components of Cooperative Scans are: a new operator in the VectorWise algebra (see Section 2.10) CScan and a new system component, the Active Buffer Manager (ABM). Figure 3.1 presents a comparison of the traditional Scan operator using the traditional buffer manager and the CScan operator using the ABM. In the traditional approach, Scan operators issue requests for I/O operations that are passed through the buffer manager that caches the data in memory according to a specified policy. The buffer manager is only responsible for handling of those requests. In particular, it is not aware of how many queries and Scan operators there are in the system.

Contrary to that, CScan operators register all needed data ranges to the ABM before starting the actual scanning. The ABM is issuing requests for I/O operations on behalf of registered CScan operators. The data is loaded in large portions (chunks) to amortize I/O latency and allow for more sophisticated chunk-level scheduling policies. Once data is loaded, the ABM feeds CScans with data. CScan operators accept data in chunk-at-time manner. Chunks can be delivered to CScans in arbitrary order. We say that CScans accept data out-of-order.

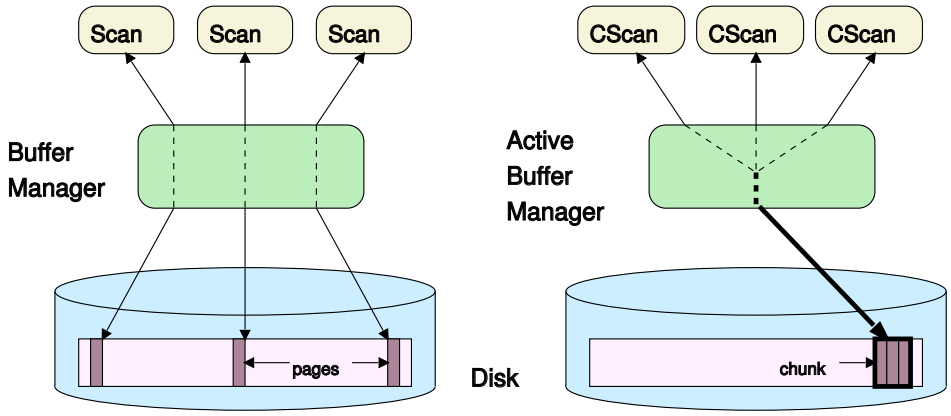


Figure 3.1: Normal scans (left) versus Cooperative Scans (right). From [ZHNB07].

3.3. The principles of Cooperative Scans

The overall goal of the ABM is to use the information provided by all registered CScan operators to exploit sharing opportunities arising in concurrent workloads, as well as to make better scheduling decisions given the flexibility of out-of-order delivery.

To allow more sophisticated scheduling policies inside ABM, we divide a table into large chunks. Their total number is one or two orders of magnitude smaller that the number of I/O units managed by the traditional buffer manager. Chunks are logical entities, they do not correspond directly to I/O units. To clarify the meaning of different entities used in the ABM and the I/O subsystem we introduce a couple of definitions that we will refer to later in this thesis.

Chunk is an area of a table containing tuples belonging to a specified continuous range. It is a logical concept that defines a horizontal slice (restriction), but does not specify which of the table columns are relevant.

Slice is a vertical slice of a chunk i.e. it is a sequence of values belonging to a certain column of a certain chunk.

Page is the smallest I/O unit handled by the VectorWise DBMS. A page contains values belonging to one column (see Section 2.4), but a page may contain data of multiple chunks and slices. This typically happens for pages containing tuples at a chunk boundary, which thus contain tuples belonging to two different chunks (and slices). But, highly compressed columns may be stored in pages that contain data of even more than two chunks and slices.

In Figure 3.2 we can can see a part of the TPC-H [Tra11] lineitem table as it is stored in the columnar storage of the VectorWise DBMS. The figure depicts chunks, slices and pages as defined above. For each column it also mentions data type (decimal, oid, str), compression scheme used (PFOR, PDICT, PFOR-DELTA [ZHNB06]) and average bits per tuple.

In a columnar storage each column is stored separately. As a result, columns differ in the amount of space they occupy either due to usage of different data types or compression. As a consequence a single page does not contain a fixed number of values, even in the same column. Therefore, in DSM we define chunks as logical concepts. A single chunk may consist of multiple physical pages and a single page may contain data belonging to multiple physical chunks.

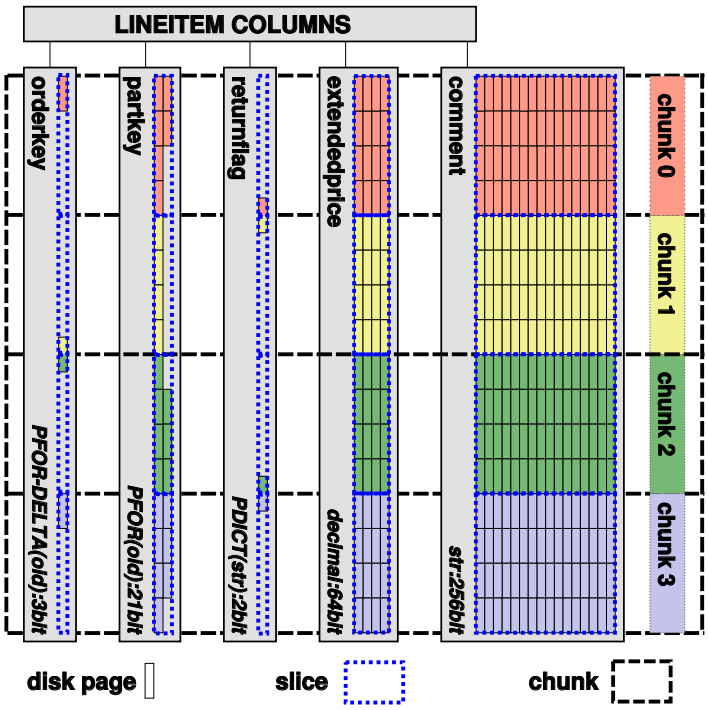


Figure 3.2: Compressed column storage (from [Zuk09])

```
1: function CscanRelevance(cscan)
2:     if not CscanStarved(cscan) then
3:         return −∞
4:     else
5:         return - ChunksNeeded(cscan) + WaitingTime(cscan) / RunnningCScans()
6:
7: function CscanStarved(cscan)
8:     return NumberOfAvailableChunks(cscan) < 2
9:
10: function UseRelevance(chunk, cscan)
11:     cols ← CscanColumns(cscan)
12:     U ← ||InterestedOverlappingCScans(chunk, cols)||
13:     P_u ← NumberCachedPages(chunk, cols)
14:     return P_u/U
15:
16: function LoadRelevance(chunk, cscan)
17:     cscan_cols ← CScanColumns(cscan)
18:     cscans ← OverlappingStarvedCScans(chunk, cscan_cols)
19:     cols ←ColumnsUsedInCScans(cscans)
20:     L ← ||cscans||
21:     P_l ← ||ColumnPagesToLoad(chunk, cols)||
22:     return L/P_l
23:
24: function KeepRelevance(chunk)
25:     starved ← AlmostStarvedCScans(chunk)
26:     cols ← ColumnsUsedInCScans(starved)
27:     E ← ||starved||
28:     P_e ← ||CachedColumnPages(chunk, cols)||
29:     return E/P_e
```

Figure 3.3: Pseudo-code of the Relevance functions

3.4. Algorithm (pseudocode)

In Cooperative Scans the CScan operator registers itself in the Active Buffer Manager. The ABM stores all metadata that is relevant for processing. It contains a list of registered CScans, a list of tables that are scanned by them and chunks that those tables are divided into. The ABM dynamically adjusts the metadata when the workload changes e.g. it registers new table when the first CScans scanning the table registers and destroys metadata related to that table when all scans scanning it leave the system.

Figure 3.4 presents the implementation of the main thread of the Active Buffer Manager. Additionally, Figure 3.3 shows a set of „relevance" functions that steer this processing. The main loop services all registered CScans making chunk loading decisions based on the current workload. It chooses a CScan for which it thinks it should load a chunk. For that CScan, the ABM finds the most relevant chunk. It subsequently loads all relevant slices for the chosen chunk. This process involves finding free space in the memory, which may result in evicting other chunks. Once the chunk is loaded, the ABM notifies all waiting queries that are interested in it.

The `ChooseCScanToProcess()` function finds a CScan with the highest value returned by the `CscanRelevance()` function. The `CscanRelevance()` function considers only starved CScans i.e. the ones with less than two chunks already cached in the memory. It prioritizes CScans with less chunks left to consume. In that way it avoids treating CScans in round-robin fashion, which has negative impact on query latency. Moreover, to avoid starving longer queries, priorities of queries are adjusted according to their their waiting time.

Once a CScan to service is chosen, the ABM has to decide which chunk to load for it. The procedure is done in the `ChooseChunkToLoad()` function by finding a chunk with the highest result of `LoadRelevance()` among the ones that are needed by the chosen query and are not fully cached. The `LoadRelevance()` aims to promote chunks that can satisfy many starved queries at low cost i.e. by performing little I/O. It calculates the number of CScans that are starved and scan a set of columns that has a non-empty intersection with the set of columns scanned by the currently serviced CScan. Also, it estimates the cost of loading a chunk by calculating how many disk pages would have to be loaded into the memory i.e. how many cold pages there are.

Loading a chunk in DSM is divided into loading all slices that are needed by the chosen CScan. Every slice may occupy a different number of pages, thus the `LoadChunk()` function finds free space in the memory for every slice separately. The `GetFreeSlot()` function makes sure that there is enough space in the memory by possibly evicting the least relevant chunk according to the `KeepReleance()`. The `KeepRelevance()` function favours chunks that are useful for many queries and occupy little memory. The `AlmostStarvedCScans()` function returns the number of starved and almost-starved CScans. The almost-starved CScans are the ones having three chunks available. We do this to avoid increasing the number of starved CScans after eviction.

Figure 3.5 presents the pseudocode of the CScan operator. The thread of CScan calls the `SelectChunk()` function when it needs to get more data to process. First, with the `ChooseAvailableChunk()` function it checks whether there are already cached chunks that can be used. If there are many of them, the one to consume is chosen according to the `UseRelevance()` function. The purpose of the `UseRelevance()` is to prioritize chunks needed by least queries, thus allowing for faster eviction of a certain chunk. In the case there are no available chunks in the memory, the query waits until it is notified by the ABM that one of requested chunk was loaded. Note, that there is no explicit mechanism telling ABM that a certain CScan is starved and waiting. CScan that hangs waiting for a chunk to be delivered

```
1: function Main
2:     while TRUE do
3:         cscan ← ChooseCScanToProcess()
4:         if cscan ← NULL then
5:             BlockForNextCScan()
6:             continue
7:         chunk ← ChooseChunkToLoad(cscan)
8:         LoadChunk(cscan, chunk)
9:         for all cscan in cscans do
10:            if ChunkInteresting(cscan, chunk) and CScanBlocked(cscan) then
11:                SignalCScan(cscan, chunk)
12:
13: function ChooseCScanToProcess
14:     relevance ← −∞
15:     chosen_cscan ← NULL
16:     for all cscan in cscans do
17:         cr ← CscanRelevance(cscan)
18:         if cscan = NULL or cr > relevance then
19:             relevance ← cr
20:             chosen_cscan ← cscan
21:     return chosen_cscan
22:
23: function ChooseChunkToLoad(cscan)
24:     chosen_chunk ← NULL
25:     L ← 0
26:     for all chunk in NeededChunks(cscan) do
27:         if not ChunkReady(chunk)) and LoadRelevance(chunk) > L then
28:             L ← LoadRelevance(chunk)
29:             chosen_chunk ← chunk
30:     return chosen_chunk
31:
32: function LoadChunk(cscan, chunk)
33:     for all column in cscan.needed_columns do
34:         num_pages ← SliceSize(cscan.table, chunk, column)
35:         FreeMemory(cscan, num_pages)
36:         LoadChunkSlice(cscan.table, chunk, column)
37:
38: function FreeMemory(cscan, num_pages)
39:     K ← 1
40:     free_pages ← FreePages()
41:     while free_pages < num_pages do
42:         for all chunk in cached_chunks do
43:             if (not CurrentlyUsed(chunk)) and (not Interesting(chunk, cscan))
44: and (not UsefulForStarvedCScan(chunk)) and KeepRelevance(chunk) < K then
45:                 K ← KeepRelevance(chunk)
46:                 chunk_evict ← chunk
47:         free_pages ← free_pages+ EvictChunk(chunk_evict)
```

Figure 3.4: Pseudo-code of the Active Buffer Manager

```
1: function SelectChunk(cscan)
2:     if Finished(cscan) then
3:         return NULL
4:     else
5:         if AbmBlocked() then
6:             SignalQueryAvailable()
7:     chunk ← ChooseAvailableChunk(cscan)
8:     if chunk = NULL then
9:         chunk ← WaitForChunk(cscan)
10:    return chunk
11: function ChooseAvailableChunk(cscan)
12:    chosen_chunk ← NULL
13:    U ← 0
14:    for all c in InterestingChunks(cscan) do
15:        if ChunkReady(c) and UseRelevance(c) > U then
16:            U ← UseRelevance(c)
17:            chosen_chunk ← c
18:    return chosen_chunk
```

Figure 3.5: Pseudo-code of the CScan operator

must be starved, thus by notifying all waiting CScans that are interested in the loaded chunk we service all starved CScans that can be satisfied at the moment.

3.5. Related work

Disk scheduling policies are part of research carried out in the area of operating systems [TP71]. The classic policies developed include First Come First Served, Shortest Seek Time First, SCAN, LOOK and others. Also, in the area of virtual memory [Bel66] and file system caching policies there were several policies introduced including LRU and MRU. It should be noted that the developed policies were aimed at access patterns and workloads different from the ones found in an analytical DBMS. Thus, the result are possibly irrelevant in the field discussed in this thesis.

In a database management system it is possible to identify several access patterns depending on the type of a query and used indices such as looping sequential, straight sequential and random [CD85]. Depending on the access pattern a different buffer management may be used. In [CD85] and subsequent work [CR93, FNS91] scans were considered trivial and handled by either LRU or MRU policy. In such case concurrently working scans were not exploiting sharing opportunities.

The idea of developing policies aimed at increasing data reuse in an environment with multiple concurrent scans was introduced in commercial DBMS systems including Teradata, RedBrick [Fer94] and Microsoft SQLServer [Coo01]. The proposed solutions used either the elevator or attach policy that were compared to Cooperative Scans in [ZHNB07].

The idea of circular scan is exploited in systems that are designed to handle concurrent workload with tens to hundreds of queries. The main purpose for using this technique is to maintain high performance of the system irrespectively of the number of concurrent queries. The Crescando system [GUM+10, UGA+09] introduces Clock Scan that services all queries working in the system. The QPipe architecture [HSA05] is another example of a system using a circular scan that feeds multiple operators with data. Moreover it supports a merge-join on two sorted relations by allowing to synchronize the merge-join operator with the current scan position. The result is generated in two batches: first the part that is shared with currently working scan is generated. Then it is followed by producing of the not-shared part. In Chapter 5 we propose a solution to merge-join processing that is more flexible. Circular

scans are also researched in [CPV09]. The CJoin operator optimizes concurrent workloads on the widely used star schema. The fact table is scanned in circular fashion to feed a pipeline of operators used to execute all active queries.

Another approach improving data reuse in multi-scan environment was introduced in IBM DB2 system [LBM+07, LBMW07]. The exploited idea is to group scans with similar speed and position together to let them reuse loaded data. The groups are controlled by allowing throttling of faster queries or recreating the groups if a considerable desynchronization occurs. In principle, the approach changes access pattern to increase locality of references, but does not change the order of processing, nor the buffer management policy.

In contrast to previous work, Cooperative Scans combine exploiting the knowledge about current workload with the flexibility of out-of-order delivery. Moreover, we present a novel approach extending the framework to support order-aware operators. Finally we discuss trade-offs related to the complexity of proposed solutions and investigate simpler ideas that provide similar benefits. Section 4.3 introduces the Predictive Buffer Manager that is a solution specially designed for workload with sequential access patterns only.

Under assumption, that full information about all future page accesses is known it is possible to formulate an optimal algorithm i.e. an algorithm that minimizes the number of pages that have to loaded from disk to the buffer. The MIN algorithm [Bel66] governs a buffer provided a sequence of all future references is available. A short proof of optimality of the MIN algorithm can be found in [Roy10]. The MIN evicts a page whose next requests occurs furthest in the future. The Predictive Buffer Manager can be seen as an approximation of the MIN that is based on current state of the system.

3.6. Summary

Existing work on Cooperative Scans [ZHNB07] convinces that this solutions offers an opportunity of high performance improvements in an analytic DBMS. However, Cooperative Scans have a significant influence on the system. Consequently, the implementation of Cooperative Scans poses a number of challenges that have to be solved before all benefits of the framework can be achieved. Chapters 4 and 5 provide a detailed description and solutions to these implementation and integration problems.

Chapter 4

Cooperative Scans in VectorWise

As discussed previously, the implementation of Cooperative Scans in the VectorWise DBMS requires several problems to be solved. This chapter describes how Cooperative Scans described in Chapter 3 are integrated into VectorWise. It addresses all problems listed in Section 2.12. Note that the problem of providing support for operators requiring ordered input stream is treated differently in this chapter and in Chapter 5. In this chapter we propose a solution reconciling regular Scan operators and Cooperative Scans, so that regular Scans are used where Cooperative Scans would lead to incorrect query results. On contrary, in Chapter 5 we provide an extension to Cooperative Scans allowing to use the framework in all query plans.

Also, this chapter introduces the Predictive Buffer Manager, a novel solution for buffer management in VectorWise.

4.1. Reconciling Cooperative Scans and regular scans

Cooperative Scans work under the assumption that the data can be delivered and processed out-of-order without influence on the correctness of query result. However, some operators work correctly only if the data is delivered fully sequentially from the underlying storage. In the VectorWise DBMS these include OrderedAggregation and MergeJoin (see Section 2.10). To introduce Cooperative Scans in VectorWise this requirement must be taken into account.

Cooperative Scans as presented in [ZHNB07] have full flexibility in terms of making chunk-level scheduling policies. All queries that ran in the system used the CScan operator, thus were willing to accept data out-of-order. As a consequence, the order of loading and evicting chunks was irrelevant as long as the system loaded one of the chunks with highest `LoadRelevance()` value and evicted one of the chunks with lowest `KeepRelevance()`. Moreover, the whole buffer space was used to cache parts of a single table. In VectorWise however, the queries scan multiple tables possibly using both CScans and traditional scanning. Therefore, we can no longer assume a dedicated buffer space for caching a single table. On the contrary, the buffer pool must cache parts of several tables.

First of all, Scan operators that comprise the query plan have to be checked for the possibility of replacing them with a CScan operator. Secondly, for every Scan operator a decision has to be made whether to replace it with CScan. Finally, the buffer space has to be shared between Cooperative Scans and regular scans. This leads to the question of a common buffer pool policy for both types of the Scan operator in VectorWise.

The following sections present present a solution to the above-mentioned problem of providing a common infrastructure for Cooperative Scans and regular scans. We discuss the way

data can be loaded in the integrated solution, as well as the eviction policy. Providing a common eviction policy for Cooperative Scans results in development of the Predictive Buffer Manager that is introduced in detail in Section 4.3.

4.1.1. Loading data

Cooperative Scans introduce a novel approach to scheduling of queries. To enable lowering the average query latency all scheduling decisions are made by the ABM thread basing on the `CscanRelevance()` function. Regular scans can not use that approach as they do not accept data delivered out-of-order. There are two possible solutions to scheduling requests for loading the data.

One idea is to keep the previous implementation of regular scans as depicted on the left side of Figure 3.1. Every scan has its own thread that issues request for I/O operations on behalf of the scan. Besides, there is the ABM's main thread that issues requests for all registered CScan operators. It is important to stress that the data for CScans is loaded on a chunk at a time basis, whereas regular scans issue requests for single pages. Scans and CScans share the same buffer space. Therefore, the system has to incorporate a common policy for making decisions on what data to evict from the buffer pool. Again, evicting data loaded for regulars scans can be done on page level, whereas for CScans on chunk or slice level.

The other, more complicated solution is to integrate regular scans with the Cooperative Scans framework. The idea is to treat regular scans as a special type of CScans, which we will refer to by sequential CScan. The sequential CScan differs from CScan by only accepting chunks in-order. Moreover, the parameters used by the scheduling policy have to adjusted. In particular, the `NumberOfAvailableChunks()` function (see Figure 3.3) has to take the current chunk that is being consumed by a sequential CScan into account. Only the chunks that directly follow the current chunk and are fully loaded for certain query can be treated as available. In this approach data for all scans working in the system is loaded by the ABM's main thread on a chunk at a time basis.

4.1.2. Evicting data

In a live system the buffer pool becomes filled with pages containing data loaded from storage. In that state, every request for loading a page that is not in the buffer pool results in evicting an already cached one. Similarly, loading a chunk will result in evicting several pages or chunks as the physical size of a chunk is not fixed. This leads to the problem of a common policy for evicting data cached in the buffer pool.

In the first solution proposed in Section 4.1.1, where the implementation of regular scans is to be unchanged, we encounter the problem of logical-physical mismatch. In particular, a request for loading a single page may result in evicting a chunk which is comprised of several pages. Similarly, a request for loading a chunk can possibly lead to evicting multiple pages and chunks. As a consequence, the regular buffer pool managing pages and the Active Buffer pool managing chunks and slices need to be integrated to make decision on which data to evict.

Now, we will focus on providing such an integrated policy for evicting data from the buffer pool. The solution leads to the idea of the Predictive Buffer Manager that can be used as stand-alone buffer manager in VectorWise, as well as integrated with Cooperative Scans. In the next section we introduce the LRU buffer manager, that is currently used as default buffer management solution in VectorWise. In Section 4.3 the Predictive Buffer Manager is introduced, whereas in Section 4.4 its integration with Cooperative Scans is discussed.

4.2. The VectorWise buffer manager

The buffer manager in VectorWise uses two priority queues governed by the Least Recently Used (LRU) policy to distinguish pages with two possible priorities: low and high. A LRU queue stores pages in order of the last use time. Most recently used pages are placed at the tail of the queue, whereas those that have not been used for longer time are closer the head of the queue. The page to evict is always popped from the head of the queue.

The policy using two LRU queue works as follows. The procedure of evicting pages tries to evict a page from the head of the low priority queue and if the queue turns out to be empty, pages from high priority queue are evicted. Thus, the pages that belong to the high priority queue are never evicted before the pages from the low priority queue. The purpose of this mechanism is to „pin" pages belonging to small tables that are frequently accessed. There are two configuration parameters that govern pinning pages. Firstly, a page is pinned if it belongs to a table containing the number of tuples under value specified in the configuration. Secondly, pages that belong to columns occupying more than a specified amount of memory are never pinned.

4.3. The Predictive Buffer Manager

In the implementation of the Predictive Buffer Manager (PBM) depicted in Figure 4.1 we generalize the idea of multiple queues. Also, we change the way priorities are assigned to the cached pages. The idea is to attempt to predict when a certain page that is cached will be accessed. Provided we have such an estimate for every page it is most reasonable to evict the one that will be accessed furthest in the future. In particular, in case all future page accesses are known the optimal solution is to evict the page that has the highest time of next usage. This algorithm is introduced in [Bel66] and proved to be optimal in [Roy10]. It is not possible to know all future queries, but it is feasible to base estimates on the current workload. The knowledge about queries working in the system at certain moment is exploited in the Predictive Buffer Manager. It is possible to estimate the time and order of future page references under the assumption that scans process data in-order. Thus, in the Predictive Buffer Manager the out-of-order delivery used in Cooperative Scans is dropped. It allows the PBM to be used as replacement for the standard LRU buffer manager in VectorWise without further modifications, as the data is delivered in the same order to the higher layers of the system.

In the Predictive Buffer Manager every page that is being loaded into the memory is assigned a priority that is based on estimated time of the next consumption by one of the scans working in the system. To enable that, every scan that enters the system calls the `RegisterScan()` function passing the table, columns and ranges to register all the pages that it is going to load. The calculation of a priority for a certain page and putting it on a appropriate queue is encapsulated in the `PagePush()` function, whereas the estimation of next consumption time is done in the `PageNextConsumption()` function.

The registering of a scan involves assigning a unique identifier to it, adding metadata to each the of pages that it will scan, as well as possible change of pages' priority. For every page we calculate how many tuples the scan will have to consume before requesting that page. This parameter is stored in the *tuples_behind* variable for each pair of Scan and page. Note that this process has to be performed separately for each scanned column, as in DSM [CK85a] pages do not hold a fixed number of values due to varying type widths and compression.

The `PageNextConsumption()` function iterates over all scans that registered need for load-

```
1: function RegisterScan(table, columns, range_list)
2:     id ← GetNewScanIdentifier()
3:     for all col in columns do
4:         tuples_behind ← 0
5:         for all range in range_list do
6:             {get a collection of pages belonging to specified column and range}
7:             pages ← GetPages(col, range)
8:             for all page in pages do
9:                 {register the scan id and the number of tuples it will have to read before consuming this page}
10:                page.consuming_scans ← page.consuming_scans ∪ (id, tuples_behind)
11:                tuples_behind ← tuples_behind + page.tuple_count
12:                {recalculate the priority of the page and push to appropriate queue}
13:                PagePush(page)
14:    return id
15:
16: function PageNextConsumption(page)
17:    nearest_consumption ← NULL
18:    for all (id, tuples_behind) in page.consuming_scans do
19:        scan ← GetScanById(id)
20:        next_consumption ← (tuples_behind − scan.tuples_consumed)/scan.speed
21:        if nearest_consumption = NULL or next_consumption < nearest_consumption then
22:            nearest_consumption ← next_consumption
23:    return nearest_consumption
24:
25: procedure PagePush(page)
26:    if page.queue ≠ NULL then
27:        PageRemove(page)
28:    nearest_consumption ← PageNextConsumption(page)
29:    if nearest_consumption = NULL then
30:        QueuePush(page, not_needed_queue)
31:    else
32:        queue_number ← nearest_consumption/TIME_STEP
33:        if queue_number ≥ Length(queues) then
34:            queue_number ← Length(queues) - 1
35:        QueuePush(page, queues[queue_number])
36:
37: procedure RefreshBmQueues
38:    tmp ← queues[0]
39:    for i ← 0 to length(queues) - 2 do
40:        queues[i] ← queues[i + 1]
41:    queues[length(queues) - 1] ← NewQueue()
42:    for all page in tmp do
43:        PagePush(page)
44:
45: procedure EvictPage
46:    if not Empty(not_needed_queue) then
47:        queue ← not_needed_queue
48:    else
49:        for i ← Length(queues)-1) downto 0 do
50:            if not Empty(queues[i]) then
51:                queue ← queues[i]
52:                brake
53:    page ← QueuePop(queue)
54:    Evict(page)
```

Figure 4.1: Implementation of the Predictive Buffer Manager

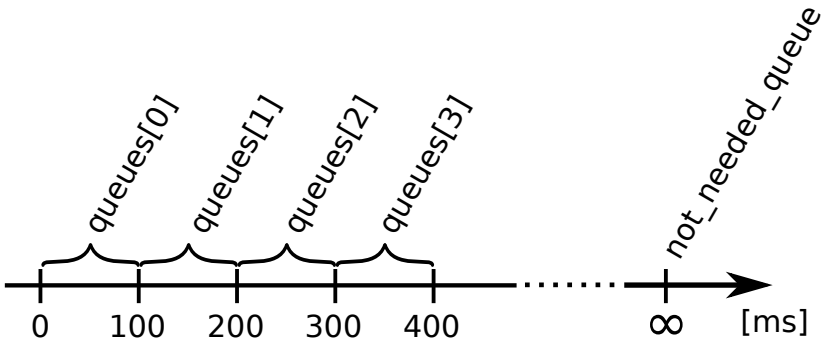


Figure 4.2: Relation between the estimated time of next consumption and priority queue in the Predictive Buffer Manager for the value of $TIME\_STEP$ equal to 100 ms

ing it. For each of them the expected time of consumption is calculated. The minimum of those is assumed to be the final result. To estimate the time a certain scan will access a specified page three parameters are needed. As discussed, the page has associated the number of tuples that the scan has to consume before reaching it. The scan periodically reports the number of tuples it already consumed, as well as its current speed of processing. Those parameters are available in *scan.tuples_consumed* and *scan.speed* variables respectively. To obtain the estimate of consumption time the number of remaining tuples to consume is divided by the speed.

The result of `PageNextConsumption()` is directly used in `PagePush()` to assign a priority to the page and push it to the corresponding queue. If a certain page will not be accessed by any of working scans it is placed on the *not_needed_queue*. Otherwise the priorities are assigned as follows. We consider a fixed time interval represented by the $TIME\_STEP$ constant and a fixed number of queues equal to `Length(`*queues*`)`. A queue with index i represents the time interval of $[i \cdot TIME\_STEP, (i+1) \cdot TIME\_STEP)$ and the last queue with the index equal to `Length(`*queues*`) - 1` represents the interval $[(\texttt{Length}(queues)-1) \cdot TIME\_STEP, \infty)$. A page is placed on a queue basing on the estimate of the next consumption. A page that is estimated to be consumed in t seconds from now falls into a queue corresponding to the time interval that contains that moment. Figure 4.2 presents how the estimate of next consumption time relates to queue pages are placed in. Note that the infinity point represents the *not_needed_queue* that stores pages that will not be consumed by any of scans working at the moment. All other pages are placed in one of $queue[i]$ queues for certain i. In particular, pages with next consumption time exceeding $(\texttt{Length}(queues) - 1) \cdot TIME\_STEP$ are put to $queues[\texttt{Length}(queues) - 1]$.

The procedure of evicting a page is similar to the one found in the previous implementation of the buffer manager in VectorWise. To evict a page we perform a search for a candidate starting from the queue of the lowest priority, continuing the search with the queues of increasing priority if a page is not found on the previous ones. Thus, the `EvictPage()` function in the Predictive Buffer Manager is similar to the policy of the LRU buffer manager with multiple priorities described in Section 4.2.

The queues need to be refreshed when the time interval of $TIME\_STEP$ passes. The process involves moving the pointers to queues by one place and recalculating priorities of pages found on the $queues[0]$. Provided the estimates are exact all the pages on the $queues[0]$ should already be consumed and pushed to the *not_neeed_queue*. If it is not the case the priority of pages from that queue is evaluated again. This procedure is performed by the `RefreshBmQueues()` function, which is run periodically every $TIME\_STEP$ seconds.

```
1: function FreeMemory(num_pages)
2:     free_pages ← FreePages()
3:     while free_pages < num_pages do
4:         elem_evict ← NULL
5:         not_needed ← not_needed_chunks ∪ not_needed_pages
6:         if not Empty(not_needed) then
7:             timestamp ← NULL
8:             for all elem in not_needed do
9:                 if timestamp = NULL or elem.timestamp < timestamp then
10:                    elem_evict ← elem
11:                    timestamp ← elem.timestamp
12:        else
13:            needed ← needed_chunks ∪ needed_pages
14:            next_consumption ← NULL
15:            for all elem in needed do
16:                if next_consumption = NULL or NextConsumption(elem) > next_consumption then
17:                    elem_evict ← elem
18:                    next_consumption ← NextConsumption(elem)
19:        free_pages ← free_pages+ Evict(elem)
20:
21: function ChunkNextConsumption(chunk)
22:     nearest_consumption ← NULL
23:     for all cscan in InterestedCScans(chunk) do
24:         next_consumption ← ChunksNeeded(cscan) /(2 · cscan.speed)
25:         if nearest_consumption = NULL or next_consumption < nearest_consumption then
26:             nearest_consumption ← next_consumption
27:     return nearest_consumption
```

Figure 4.3: Integrated eviction policy for the Cooperative Scans and regular scans

Another situation, where a page is assigned a new priority is its consumption. When a query that requested the page finished processing of it the `PagePush()` function is called again to recalculate the priority. In particular, if there are no other queries requesting that page, it is moved to the *not_neeed_queue*.

4.4. Integrating the Predictive Buffer Manager with the Cooperative Scans

Section 4.3 discussed the implementation of the Predictive Buffer Manager in the VectorWise DBMS. The presented solution can be used as stand-alone buffer manager or integrated with Cooperative Scans. As outlined in Section 4.1.1 and 4.1.2, one possible solution for integrating regular Scan operators with CScans is to provide a common policy for evicting pages. The Predictive Buffer Manager provides metadata for determining the time of next consumption for each cached page. This metadata is used to compare the priority of pages that were loaded with regular Scans and chunks loaded by the ABM.

The integrated eviction policy is presented in Figure 4.3, where the `FreeMemory()` function from Figure 3.4 is reimplemented. It handles both chunks and single pages cached by the buffer. First, pages or chunks that are not marked as needed by any of running CScan/Scan operators are considered to be evicted. If there are such pages or chunks, the one to evict is based on its *timestamp* parameter. The *timestamp* signifies the time a page or a chunk was last consumed. Effectively, the not needed pages and chunks are managed by a LRU policy.

In case all cached pages and chunks are needed by one of running queries, the procedure has to evict one of them. The procedure chooses the one with highest estimate of next consumption time. The `NextConsumption()` function handles both pages and chunks by calling `PageNextConsumption()` (presented in Figure 4.1) or `ChunkNextConsumption()` (presented

in Figure 4.3) appropriately. The estimation when a certain chunk will be consumed is complicated due to the dynamic nature of scheduling in Cooperative Scans. Thus, we provide a rough estimate that is based on the length and the speed of a CScan. Suppose a CScan requesting 10 chunks is processing data at pace of 1 chunk a second. We do not know the order in which chunks will be consumed, but on average a chunk will be consumed after 5 seconds. In the `ChunkNextConsumption()` we find the CScan operator that is expected to consume a certain chunk first. The expected effect of this approach is eviction of chunks that are needed only by long and/or slow CScans, so that they will not be consumed soon.

Alternatively, it would be possible to use the `UseRelevance()` score to determine in what order already-cached chunks may be consumed. After determining the order, the time of consumption could be calculated using the current speed of processing of a certain CSccan.

4.5. Handling of PDT-resident updates

The VectorWise DBMS uses special in-memory differential structures called Positional Delta Trees (PDTs) [HZN+10] to handle modifications of a database. PDTs are introduced and described in Section 2.8. The Scan operator performs a PDT merging operation to produce output containing all applied changes. For correctness of query results the CScan operator needs to support PDT merging as well. The main difference between the Scan and CScan operator is the fact that the Scan operator delivers data in-order, whereas the CScan operator delivers data out-of-order in chunk-at-time manner. The used method influences the PDT merging operation. Below, we discuss implementation details of the support for updates in the CScan operator.

The updates stored in PDTs are indexed by position where they have to be applied. Consequently, in the VectorWise DBMS two types of a tuple identifier are distinguished. The Stable ID (SID) is a 0-based dense sequence identifying tuples stored in the stable storage i.e. the state of a table without updates applied. The Row ID (RID) is a 0-based dense sequence enumerating the stream that is visible to the query processing layer i.e. after updates are applied to data fetched from the stable storage. The RID is generated during PDT merging process. The SID and RID sequences are different unless all PDTs are empty i.e. PDT merging is an identity operation.

It is viable to provide a mechanism to translate between SID and RID. Let us focus on RID to SID translation first. For tuples belonging to the stable storage (stable tuples) and not marked as deleted in PDTs, the RID that is assigned to that tuple can be trivially translated to its SID, as every stable tuple has it unique SID. Also, for tuples that are not part of the stable storage i.e. they are stored in PDT structures it is possible to translate RID to SID. Those tuples are assigned SID of the first stable tuple that follows it.

As a consequence, in case new tuples are inserted and stored in PDTs only, there may be multiple tuples that are assigned the same value of SID. Thus, it is not possible to define SID to RID conversion as an inverse of RID to SID conversion, because the RID to SID conversion is not an injective function. However, it is possible to introduce two possible variants of SID to RID conversion. We can either translate a certain SID into lowest RID that maps to that SID, or the highest one.

In VectorWise PDT structures provide an interface to perform all types of above-mentioned conversions: `RIDtoSID()` for the RID to SID conversion, `SIDtoRIDlow()` for the „low" variant of SID to RID conversion and `SIDtoRIDhigh()` for the „high" variant of SID to RID conversion.

Figure 4.4 depicts an example of conversion between SID and RID. Below, we can see stable tuples stored on the hard disk and their respective SIDs. After tuples are loaded and merged

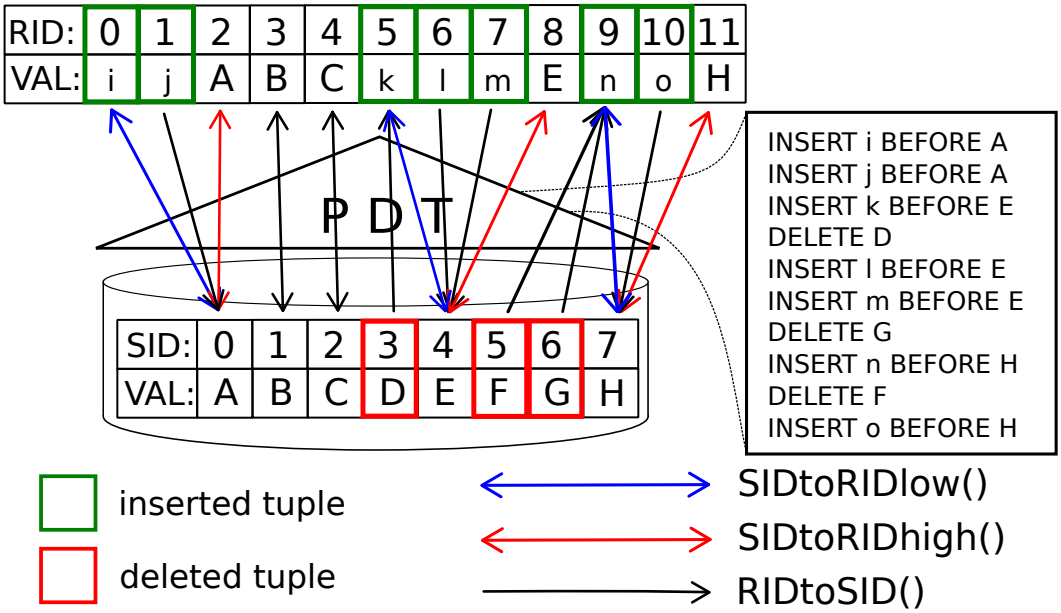


Figure 4.4: Conversion between SID and RID

```
1: function RegisterCScanRanges(table, rid_ranges)
2:     sid_ranges ← ∅
3:     for all rid_range in ranges do
4:         sid_range ← RIDtoSID(cscan.pdt, table, rid_range)
5:         sid_ranges ← sid_ranges ∪ sid_range
6:
7: function CScanNext
8:     if Finished(current_range) then
9:         if not empty(cscan.ranges_in_current_chunk) then
10:            cscan.current_range ← pop(cscan.ranges_in_current_chunk)
11:            SetRidRange(cscan, cscan.current_range)
12:        else
13:            chunk ← SelectChunk(cscan)
14:            chunk_rid_lo ← SIDtoRIDlow(cscan.pdt, cscan.table, chunk.sid_lo)
15:            chunk_rid_hi ← SIDtoRIDhigh(cscan.pdt, cscan.table, chunk.sid_hi)
16:            chunk_rid_range ← [chunk_rid_lo, chunk_rid_hi]
17:            overlapping_ranges ← FindOverlappingRanges(cscan.ranges, chunk_rid_range)
18:            cscan.ranges_in_current_chunk ← ∅
19:            for all range in overlapping_ranges do
20:                cscan.ranges ← cscan.ranges\range
21:                (overlap, rest) ← range_trim(range, chunk_rid_range)
22:                if not empty(overlap) then
23:                    cscan.ranges_in_current_chunk ← cscan.ranges_in_current_chunk ∪ overlap
24:                    if not empty(rest) then
25:                        cscan.ranges ← cscan.ranges ∪ rest
26:                cscan.current_range ← pop(cscan.ranges_in_current_chunk)
27:                SetRidRange(cscan, cscan.current_range)
28:    ⋮                                                    ▷ Further processing
```

Figure 4.5: Implementation of support for updates in the ABM and CScan operator

with changes stored in PDTs, we obtain a new stream of tuples that is enumerated with RID. Deleted tuples are marked with red, whereas the inserted ones with green. Arrows indicate the translation between SID and RID. The blue arrows indicate results of the `SIDtoRIDlow()` function, whereas the red ones correspond to the `SIDtoRIDhigh()`. Note that some of the arrows are only one-way. They indicate tuples, for which it is not possible to reverse the RID to SID conversion, because they are in the middle of a sequence of tuples with the same value of SID. The deleted tuples are stored on the hard disk and loaded into the memory but omitted in the PDT merging process. Thus, there is no RID that translates into SID of those tuples. However, it is still possible to translate their SIDs to RIDs. The assigned RID is the lowest RID that translates into a SID higher than the one of the deleted tuple.

The Active Buffer Manager has been designed to work purely on the storage level. It is not aware of existence of in-memory updates and PDTs in particular. Thus, the ABM has to be notified about data ranges expressed in SID i.e. RID ranges requested by the query processing layer have to be translated into SID ranges. This process is performed in the `RegisterCScanRanges()` function presented in Figure 4.5. The CScan operator is passed a list of RID ranges that it has to produce to its parent operator. Those RID ranges are converted into SID ranges that the ABM operates on. From now on, the ABM knows which stable tuples have to be loaded.

Once the ABM loads a chunk and passes it to the CScan operator, an inverse process has to be performed. As the RID to SID translation is not reversible the algorithm is more complicated. Given a certain chunk, which is effectively a range of stable tuples that are ready in the memory, we can determine what RID range can be generated out of them. The process is implemented in the `CScanNext()` function as presented in Figure 4.5.

The `CScanNext()` function implements the `next()` method of the CScan operator (see Section 2.3). It is called by the parent operator to request more tuples to be fetched. As discussed in Section 2.10.1 the Scan operator (and also CScan) scans a list of specified ranges. With every Scan we can associate a range that it is processing at the moment. The `next()` method either outputs a vector of tuples in the current range or when it is finished, advances to the next one. The `CScanNext()` function first checks whether the current range, that is stored in the *cscan.current_range* variable, has been completely passed to the parent operator. If so, it tries to get a new range from the *cscan.ranges_in_current_chunk* which contains all ranges that can be generated with the current chunk that is provided by the ABM.

If *cscan.ranges_in_current_chunk* is empty a new chunk has to be requested. To do it, `CScanNext()` calls the `SelectChunk()` function described in Section 3.4. Once a chunk is ready to use, the CScan operator has to determine what RID ranges can be generated with stable data of that chunk. First, the maximal RID range that corresponds to the SID range of current chunk is determined. It is represented by *chunk_rid_range*. Next, ranges belonging to the *cscan.ranges* collection, that overlap with the *chunk_rid_range* are found to generate all ranges that can be processed fully or partially by using the current chunk. The *cscan.ranges* collection contains all remaining ranges that still have to be generated by the CScan operator. The aim is to create new contents of the *cscan.ranges_in_current_chunk* collection and put ranges that will have to generated with other chunks in *cscan.ranges*. To find out the overlapping and non-overlapping part the `RangeTrim()` function is used. It accepts two ranges and returns two results: the overlapping part and the second argument subtracted from the first one. The result is stored as a pair $(overlap, rest)$, where *overlap* is the part of a range that can be processed with the current chunk, whereas *rest* belongs to another chunk. Both *overlap* and *rest*, if not empty, are inserted into *cscan.ranges_in_current_chunk* and *cscan.ranges* collections respectively.

When a range to process is finally determined, the `SetRidRange()` function is called. It

```
Append(table, <data stream 1>)
Scan(table)
```
(a) Transaction 1 (T1)

```
Append(table, <data stream 2>)
COMMIT
Scan(table)
```
(b) Transaction 2 (T2)

```
Append(table, <data stream 3>)
Scan(table)
```
(c) Transaction 3 (T3)

```
Append(table, <data stream 4>)
Scan(table)
```
(d) Transaction 4 (T4)

Figure 4.6: Transactions appending to a table

is responsible for setting the state of PDTs and the CScan operator to appropriately to start the PDT merging process in the specified RID range. The PDT merging does not need to be modified in comparison to the Scan operator.

4.6. Handling of concurrent appends

The VectorWise DBMS implements the snapshot isolation [BBG+95] approach to handle concurrent transactions as described in Section 2.7. Every transaction that runs in the system works on a local snapshot of the database. The implementation of snapshot isolation in VectorWise is based on the Positional Delta Trees (PDTs) differential structures [HZN+10]. As explained in Section 2.8, PDTs form a hierarchy with global and large PDTs at the bottom and small ones at the top. The top-most PDTs are local to a transaction i.e. contain only updates visible to its transaction. The tables stored on disk remain unchanged, since all modifications are stored in memory-resident PDTs. Therefore, in most cases the Active Buffer Manager does not need to handle different snapshots that transactions work on separately, as it works on the storage level and is not aware of updates stored in PDTs. To provide support for updates stored in PDTs the CScan operator has to perform the PDT merging process as described in Section 4.5.

In VectorWise a snapshot of each table is a memory-resident set of arrays that contain references to pages (page identifiers) that belong to each column of the table (one array per column). Such snapshot is not changed when modifications are stored in PDTs. However, as described in Section 2.10.10, the Append operator adds new tuples to the table by appending new pages with data to the snapshot of the transaction that executed the Append.

Figure 4.6 presents simple transactions that perform an Append and scan the table afterwards. Transaction `T2` differs from the other by using the `COMMIT` statement to make the changes it made persistent. Suppose `T3` and `T4` start after `T2` ends and `T1` starts before `T2`'s end. An example course of execution of those transactions and snapshots that they are working on is presented in Figure 4.7. For simplicity we assume that the table consists of a single column. Snapshot 1 represents the initial state of the table. The table consists of four pages identified with numbers from 0 to 3. It is the master snapshot, that all transactions start with, thus it is marked with red. The `T1` and `T2` transactions begin with appending new data to the table. As a result, two different transaction-local (marked with blue) snapshots are created: Snapshot 2 and Snapshot 3. They share references to the first four pages and contain different last two pages that were created separately for each transaction by the Append operator. The `T2` commits applied changes and `T1` does not. Consequently, the snapshot that `T2` worked on becomes the master snapshot (marked with red). From now on, all new transactions will use Snapshot 3. Thus, transactions `T3` and `T4` append data to the new snapshot. Note, that the

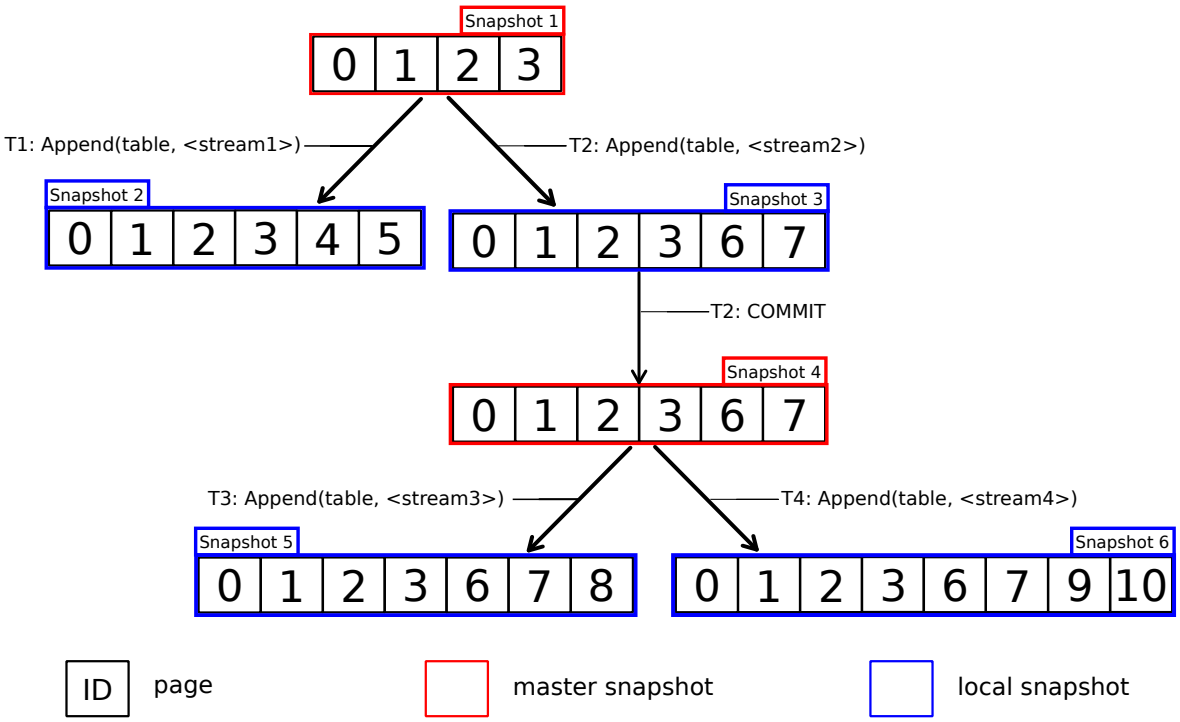


Figure 4.7: An example of how snapshot isolation works in VectorWise

`T1` transaction may still work in the system scanning pages belonging to Snapshot 1.

The crucial observation is the fact that all transactions running in the system work on snapshots that have a common prefix. In the above-mentioned example the common prefix consists of pages {0, 1, 2, 3} before the transaction `T2` commits. When `T1`, `T3` and `T4` work at the same time the common prefix is {0, 1, 2, 3} as well, whereas when only `T3` and `T4` are executed the common prefix is the set {0, 1, 2, 3, 6, 7}.

In the presented scenario the Active Buffer manager needs to detect transactions working on different snapshots. The fact that snapshots have common prefixes can be exploited, so that queries can benefit from sharing opportunities regardless of differences in snapshots. To do that, we introduce in the ABM a notion of shared chunks and local chunks. Shared chunks consist of pages that belong to two or more snapshots that are used by transactions at certain moment. Pages of local chunks belong to only one snapshot. It should be noted that chunks encompass multiple columns. Thus, a chunk can be regarded as shared only if all its pages in all columns belong to snapshots of two transactions. In particular, even after appending a single value to a table, its last chunk becomes local.

Figure 4.7 can be also interpreted on chunk level instead of page level. This is to some extend a simplification, because as stated before Appends make the last chunk local. However, this does not change the logic that handles concurrent Appends and the overall idea. Let us focus on such a situation from perspective of the ABM. Shared chunks can be detected by finding the longest prefix that belongs to at least two snapshots. Suppose, transactions `T1`, `T3` and `T4` are working the system. In this case the longest shared prefix is the set {0, 1, 2, 3, 6, 7}. Consequently, we exploit that knowledge and load those chunks earlier as it increases sharing opportunities.

Note, that the longest shared prefix can change as queries enter and leave the system. For example, at the time `T1` and `T2` are the only working transactions the shared prefix is comprised of chunks {0, 1, 2, 3}. It is also the case for `T1`, `T2` and `T3` being executed. However,

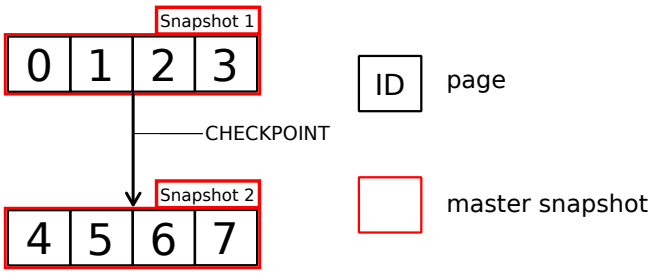


Figure 4.8: Behaviour of a checkpoint. PDT changes are applied to create a new snapshot.

when `T2`, `T3` and `T4` work in the system the longest prefix that can be found is the set {0, 1, 2, 3, 6, 7}. An important observation is the fact that the longest shared prefix is in fact the only shared prefix that can be found. Indeed, suppose that at certain moment there are two transactions whose snapshots contain pages {0, 1, 2, 3, 4, 5} (Snapshot 2 in Figure 4.7) and two transactions working on pages {0, 1, 2, 3, 6, 7} (Snapshot 3 in Figure 4.7). Having two transactions working on identical snapshots means that this snapshot must have been the master snapshot at the moment these transactions started, or one transaction already committed (making its snapshot the master snapshot) before the other started. Thus, both Snapshot 2 and Snaphost 3 must have been master snapshots at some moment. This is impossible, because they have a non-empty intersection, but none of them is a subset of another i.e. none of them could have been created by appending to another. In VectorWise only one of concurrent transactions that applied modifications to its snapshot can commit its changes and make its snapshot the master snapshot. The other transactions are detected to be conflicting and consequently aborted. Thus, there may be concurrent transactions working either on Snapshot 2 or Snaphost 3, but not on both.

Every time a new CScan scanning a certain table enters the system the ABM finds the longest prefix that is shared by at least two CScan operators and marks chunks accordingly. Chunks belonging to the found prefix are marked as shared, whereas the other ones are marked as local. The same procedure is performed when a CScan leaves the system as it may influence the longest shared prefix as well. Keeping track of shared and local chunks allows to maximize sharing opportunities in case transactions work on similar, but not equal versions of the same table. Moreover, it allows proper adjusting of matadata used for scheduling (see Section 3.4). In particular, shared chunks have higher changes to be loaded and kept in the memory longer, whereas the local chunks need to be loaded and used once, possibly in the final phase of a certain CScan.

4.7. Handling of checkpoints

Figure 4.8 presents working of a checkpoint in the VectorWise DBMS. As discussed in Section 2.9, a checkpoint involves creating a new version of the same table that contains all changes stored in Positional Delta Trees. After checkpoint is finished, the checkpointed table is stored in newly-allocated pages and the PDTs can be flushed. As depicted in Figure 4.8 a checkpoint results in creating a new master snapshot (marked as red) that consists of new pages. After the checkpoint is finished and the new version of the master snapshot created. All starting transactions will use Snapshot 2. At the same time, there may be still transactions in the system working on the old Snapshot 1. Once all transaction using Snapshot 1 are finished, the Snapshot 1 can be destroyed and pages belonging to it marked as available to be reused.

The ABM needs to detect that a certain table has been checkpointed, as transactions using Snapshot 1 and Snapshot 2 work on non-overlapping sets of pages, thus can not share pages loaded in the buffer pool[1]. Every time a new CScan is registered, the ABM verifies the set of pages belonging to the snapshot of transaction that CScan is part of. There are four possible cases to handle:

(i) It is the first CScan that accesses that particular table – the ABM has to create and initialize metadata for chunks and slices i.e. register new chunks and slices.

(ii) There are other CScans working on the same table and their snapshot is identical – the ABM does not need to change its metatada related to tables.

(iii) There are other CScans working on the same table and a common prefix with other CScans' snapshots can be found – the ABM has to find shared and local chunks as discussed in Section 4.6.

(iv) There are other CScans working on the same table and the snapshot of the new CScan contains different pages that the other snapshot – the ABM registers new version of the same table along with its chunks (as in (i)).

The metadata in the ABM needs to reflect changes in the database after checkpoint was finished. Thus, every time a CScan operator finishes and unregisters itself, the ABM verifies whether there are other CScans working on a snapshot that is identical or has a common prefix with the snapshot of the leaving CScan. In case there are no such CScans the metadata related to chunks belonging to that version of the table is destroyed.

4.8. Cooperative Scans and parallelism

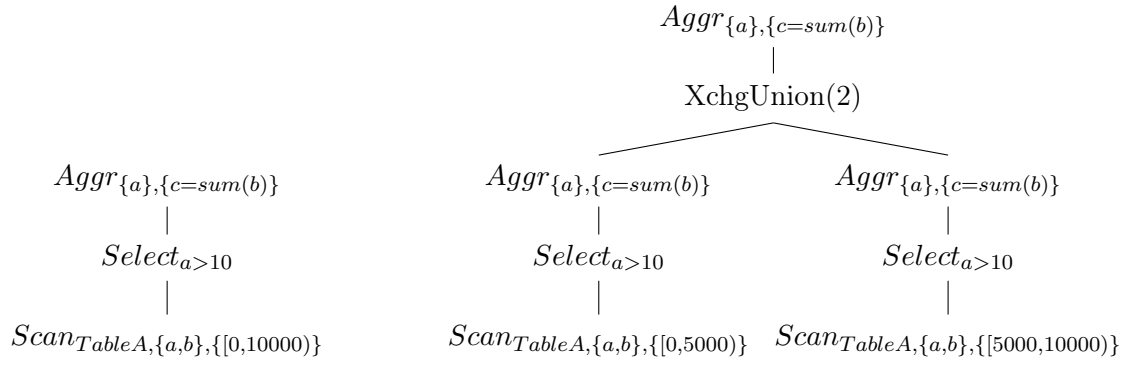
The VectorWise DBMS employs intra-query parallelism that is implemented by means of a set of Xchange (Xchg) operators [Gra90]. A query plan of a parallelized query is transformed to contain multiple copies of certain subtrees that are connected with special Xchange operators. Those subtrees are executed in parallel by multiple threads. An example of such a transformation is depicted in Figure 4.9.

The subtree under the Aggr operator has been duplicated and connected with the XchgUnion operator that merges two streams into one. On top, there is another Aggr operator added. In particular, the Scan operator has been split into two separate operators that scan parts of the original range. In the VectorWise DBMS ranges are split equally between working threads and assigned to each of Scans as presented in Equation 4.1.

$$\text{range } [a..b) = \begin{cases} \text{range } [a \,..\, a \;+ \frac{(b-a)*1}{n}) & \text{for the first Scan} \\ \ldots & \ldots \\ \text{range } [a + \frac{(b-a)*(n-1)}{n} .. a \;+ \frac{(b-a)*n}{n}) & \text{for the } n\text{-th Scan} \end{cases} \tag{4.1}$$

If Cooperative Scans are enabled parallelization works in the same way. The only difference is that Scan operators are replaced with CScan operators. It has several consequences for the Cooperative Scans framework. First of all, there are multiple CScan registrations in the Active Buffer Manager instead of one. In the current implementation ABM is not aware that

[1] In principle, there are still opportunities at the logical level, but we decided to ignore them in out solution, as it is a very rare case.



(a) Sequential query plan (b) Parallel query plan

Figure 4.9: An example of a transformation of a sequential query plan into a parallel query plan

those CScan operators belong to the same parallel query plan. One implication is a possible difference in scheduling of loading data. In most cases, CScans belonging to the same parallel query plan request the same number of chunks and process them at similar pace. However, if one or more threads turns out to be much faster than the others, the ABM will start treating one of CScans as shorter, thus prioritize it. Unfortunately, it can have negative impact on execution time of the whole query. If the ABM choses to prioritize a subset of CScans, the others may be waiting for data longer. As a result, the system fails to exploit speed-up opportunities of parallel execution. Secondly, the usage of CScans in parallelized query plans may lead to increased data skew. Regardless of the actual range to be scanned, CScan operators process data in chunk-at-time manner and have to wait until full chunk is loaded (the chunk range is trimmed as described in Section 4.5). A CScan, whose range encompasses few chunks has to load more data than really needed. This problem becomes more relevant after a single CScan is divided into multiple CScans scanning smaller ranges.

Another problem is the static division of ranges. In most cases, the threads processing data in a parallelized plan do not need to be bound to specific data ranges. They accept any tuples, as long as they fall into the original range of the sequential plan. With this approach the ability of CScans to accept data out-of-order is not fully exploited.

Suppose the parallelization generates a plan with two CScan operators scanning chunks from 0 to 4 and from 5 to 9 respectively. Let us assume that the buffer pool contains a set of the following chunks: {3,5,6,7,9}. As we can see, there is one chunk available for the first CScan and four available for the second one. Thus, in the current implementation the second CScan will only have to wait for loading chunk 8, whereas the first one needs to load. Consequently, the first CScan has to wait for much more I/O to be performed delaying the execution of the whole query. A more optimal approach would be to distribute the already-loaded chunks evenly between participating CScans and also distribute evenly the newly loaded chunks. This problems leads to idea of the Cooperative Xchange Scan that is discussed further in Section 8.2.2.

4.9. Introducing Cooperative Scans in query plans

Every query that enters the kernel of the VectorWise DBMS is expressed in the VectorWise algebra (see Section 2.10). The query plan is expressed as a tree of operators. In all cases

```
1: function RewriteCScan(root, node)
2:     needs_order ← GetProperty(node, PROP_ORDER)
3:     if node.type = Scan and needs_order = TRUE and StreamSize(node) > THRESHOLD then
4:         NodeReplace(node, CScan)
5:     else if node.type = MergeJoin or node.type = OrdAggr then
6:         PropagateOrderProperty(node, TRUE)
7:     else if node.type ∈ OrderDestroyingOperators then
8:         PropagateOrderProperty(node, FALSE)
9:     else if node.type = HashJoin then
10:        left_child_needs_order ← GetProperty(node.left_child, PROP_ORDER)
11:        SetProperty(node.left_child, PROP_ORDER, (needs_order or left_child_needs_order))
12:    else
13:        PropagateOrderProperty(node, needs_order)
14:    for all child in node.children do
15:        RewriteCScan(root, child)
16:
17: function PropagateOrderProperty(node, value)
18:    for all child in node.children do
19:        needs_order ← GetProperty(child, PROP_ORDER)
20:        SetProperty(child, PROP_ORDER, (needs_order or value))
```

Figure 4.10: Rewriter rules introducing CScan operators in query plans

the leafs of such a tree are the Scan operators that retrieve data from the storage and pass it to parent operators for processing. Query plans can be transformed by the Rewriter module (see Section 2.11) that can modify them in various ways. In the case of Cooperative Scans, queries that will use the framework also have to be transformed with the Rewriter. In fact, it only has to replace Scan operators with CScan operator that register in the Active Buffer Manager. Thus, the shape of the tree remains unchanged.
To introduce a CScan operator in the place of Scan operator two conditions have to be met:

(i) the query plan should return the same result if data is delivered out-of-order

(ii) the size of scanned relation needs to be big enough, so that it can be divided into several chunks and registered in the Active Buffer Manager

The procedure of replacing Scan operator with CScan equivalents is implemented by adding an additional procedure to the Rewriter module. The `RewriteCScan()` function presented in Figure 4.10 performs both appropriate annotating of the nodes and rewriting the query plans to use CScans where applicable. The $PROP\_ORDER$ property determines whether a certain Scan can be transformed into a CScan. It is set to $FALSE$ when a certain node can deliver data out-of-order to its parent operator. We assume that at the moment the `RewriteCScan()` function is executed the value of this property equals $FALSE$ for all nodes. The underlying idea is that we assume the whole query plan to accept data delivered out-of-order, and then find nodes, for which this assumption can not be held.

The `RewriteCScan()` function traverses the tree in top-down fashion. At each node it checks what operator it represents. If a Scan operator has been encountered, it verifies the value of the $PROP\_ORDER$ property and checks whether the size of the table exceeds a predefined threshold. If both conditions are met a Scan can be replaced with a CScan. In case an operator that requires the arriving stream to be ordered or clustered (MergeJoin or OrderedAggregation) is encountered, all its child operators are notified that the stream they deliver needs to be ordered by setting their $PROP\_ORDER$ property to $TRUE$. Similarly, when an operator that changes the order of the stream is found, the $PROP\_ORDER$ is set to $FALSE$. The operators that change the order of incoming stream include Sort, TopN and Aggregation. Those operators accept unordered data streams. The

`PropagateOrderProperty()` function is a helper function that performs the propagation to child operators. It sets the value of $PROP\_ORDER$ to the Boolean sum of the current value and the propagated value. The reason for such implementation is the fact that certain nodes may have more that one parent nodes (see Section 2.10.9). Thus, it has to detect cases where one of the parent operators needs ordered stream and the other does not. Another case that has to be treated separately is the HashJoin operator. Here, we only propagate the $PROP\_ORDER$ property to its left child omitting the right child. The stream arriving on the right side of a HashJoin does not need to be ordered (see Section 2.10.5), thus we do not propagate the eventual requirement of ordering. Finally, if none of above-mentioned cases is identified, we propagate the value of $PROP\_ORDER$ to all node's children.

4.10. Summary

This chapter provided further implementation details of Cooperative Scans in VectorWise and addressed problems stated in Section 2.12. The work described in this chapter provides a fully-functional implementation of Cooperative Scans that can be incorporated into VectorWise. In particular to solve the problem of support for order-aware operators, we propose to introduce an ability to use traditional scan processing in query plans where it is not possible to use Cooperative Scans. To do that, we integrate Cooperative Scans with traditional scan processing. The development of integration of these to approaches led to creation of the Predictive Buffer Manager (PBM) that can be used as a standalone buffer manager in VectorWise.

We emphasize that the above-mentioned approach may miss sharing opportunities, as traditional scan processing processes data in-order and is not able to share data loaded by other scans in different positions of a table. We address this issue in Chapter 5 by developing an extension of Cooperative Scans that enables out-of-order delivery for all operators found in the VectorWise algebra (see Section 2.10).

Chapter 5

Cooperative Merge Join

This chapter introduces a novel algorithm, the Cooperative Merge Join (CMJ) that is aimed at extending Cooperative Scans with support for order-aware operators. Furthermore, we show how introduction of the Cooperative Merge Join influences the implementation of Cooperative Scans and revisit some of aspects already discussed in Chapter 3 and 4.

5.1. Motivation

The Cooperative Scans framework introduces the notion of out-of-order delivery. The VectorWise DBMS system uses a pipelined query execution model, where a query is represented as a tree of operators (see Section 2.3). Those operators encapsulate algorithms that are performed on a stream of data in a vector-at-time fashion. Some of those operators require the incoming data stream to be ordered, thus can not work properly with the Cooperative Scans, as the CScan operators deliver data out-of-order. A description of the most important operators in VectorWise can be found in Section 2.10. The two main order-aware operators in VectorWise are OrderedAggregation and MergeJoin. In this chapter we propose the Cooperative Merge Join, an extension to Cooperative Scans providing support for both order-aware and unaware operators. By doing so we allow all operators to benefit from exploiting of sharing opportunities that is provided by the Cooperative Scans framework.

5.2. Partitioning of sorted and indexed tables

[ZHNB07] proposes a modification of the OrderedAggregation operator that accepts data arriving in chunks that are internally sorted, but not globally. The proposed modification is aimed at exploiting the fact of internal sorting of chunks. The aggregation is performed in the same way for the internal key values of every chunk. The first and the last key values in every chunk are stored till the final result can be calculated after arrival of neighboring chunks.

We chose to follow a simpler approach that does not require a modification of the OrderedAggregation operator. To avoid saving boundary values, we propose to perform partitioning of a table into chunks, such that two consecutive chunks do not contain tuples with the same key value.

The same approach can be extended to support the MergeJoin operator (see Section 2.10.6). The partitioning has to be aligned to key boundaries on both sides. Moreover, the boundaries of chunks with the same number have to be aligned to the same key boundary on both sides. An example of such partitioning is depicted in Figure 5.1. The Child table is sorted on the `FK` column. `FK` is a foreign key (see Section 2.1.1) that references the `ID` column of the Parent

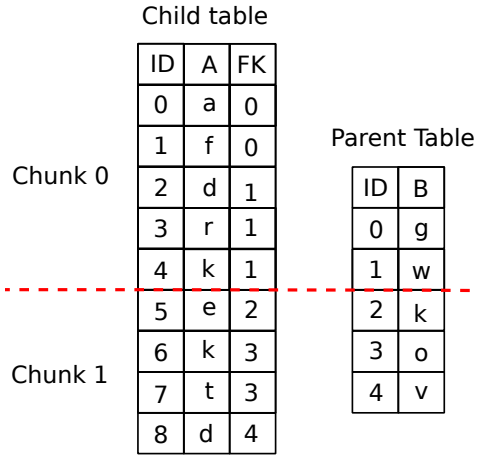


Figure 5.1: Example partitioning of two clustered tables

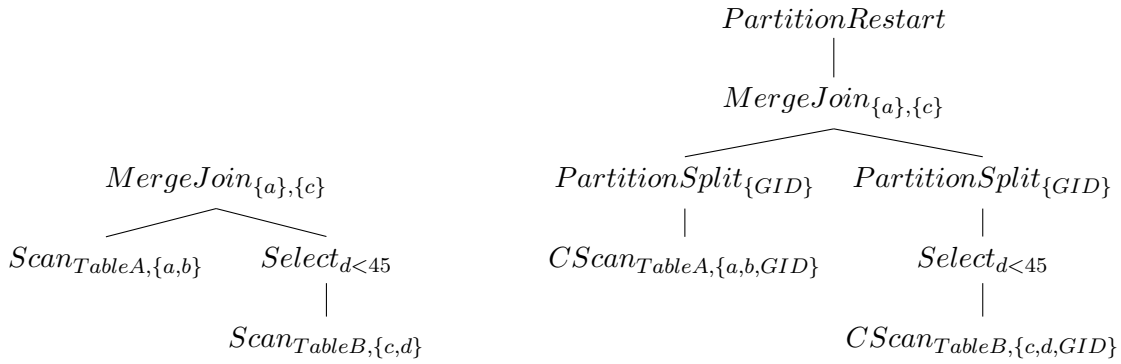
table. In case two tables are connected with a foreign key, VectorWise can employ join indices (see Section 2.6). Figure 5.1 presents a scenario, where a join index is created i.e the Child table is clustered on the `ID` column of the Parent table. Both tables are divided into two chunks. On the child side chunk boundaries are aligned to key boundaries of the `FK` column, whereas on the parent side it is aligned to the same key boundaries of the `ID` column.

The ABM needs to create such partitioning once the first query operating on a certain pair of clustered tables enters the system. The boundaries of chunks are based on entries in the Join Index Summary structures that defines partitioning of both clustered relations that is aligned as described above. The ABM chooses which boundaries defined in the JIS to use as boundaries of chunks based on the size of chunk that is specified in the configuration. It is not possible to obtain chunks of the same size that matches the target size. The accuracy of division depends on the number of entries in the JIS. Consequently, the granularity of partitioning in the JIS needs to be equal or finer than the partitioning into chunks.

5.3. Order-aware operators and CScans in query plans

In the VectorWise DBMS the OrderedAggregation (see Section 2.10.4) operator accepts a stream that is clustered on the aggregation key. Such stream can be regarded as a sequence of distinct key values, where each value may be repeated several times. The exact order of key values is not important from the perspective of the OrderedAggregation operator. Thus, the values can be reordered, as long as all repetitions of the same value are never separated by a different value. As described in Section 5.2, such a stream is divided into chunks, such that there are no two consecutive chunks containing the same key value. Reordering of chunks preserves the clustering property. Consequently, query plans that use the OrderedAggregation do not need to be changed when a regular Scan is replaced with CScan, as long as chunk boundaries are aligned appropriately.

The MergeJoin operator in VectorWise assumes both input streams to be globally sorted on the key attribute. Thus, data arriving out-of-order in chunk-at-time manner breaks that property. We achieve proper functioning of the MergeJoin operator with Cooperative Scans by using sandwich operators described in Section 2.10.11. MergeJoin is sandwiched between PartitionRestart and two instances of PartitionSplit. Figure 5.2 (a) depict the original QET with a merge-join of two relations, whereas Figure 5.2 (b) show how the MergeJoin operator is sandwiched. The scheme using a query plan with a sandwiched MergeJoin fed with data



(a) Original merge-join query plan

(b) Query plan using Cooperative Merge Join

Figure 5.2: A simple query plan using merge-join and its transformation to a query plan using the Cooperative Merge Join

by CScan operators will be called the Cooperative Merge Join (CMJ).

PartitionSplit operators are responsible for detecting partition (chunk) boundaries in the incoming data stream. To do that, an additional virtual GID column is added by the CScan operators to both data streams. The GID column signifies the chunk number that is currently being outputted by the CScan operator. In particular, the GID of the first chunk always equals 0.

Chunks arriving at the MergeJoin operator have to be synchronized i.e. should contain data with the same range of key values. The synchronization is done in the Active Buffer Manager, which is aware of pairs of CScan operators that take part in the Cooperative Merge Join. The ABM keeps track of a pair of chunks that are being processed by both sides. When a new chunk is requested by one of CScan operators, the ABM first checks, whether the other CScan participating in the CMJ already received its chunk. If so, the requesting CScan has to synchronize with the one that already has a chunk assigned. The matching chunk is found by the ABM and returned (and possibly loaded in case it was not available in the memory). In the other case, if none of CScans are assigned a chunk to output, the ABM has flexibility to make a choice which chunk to assign. This is done using a modified `UseRelevance()` function described in Section 5.4.

The PartitionRestart operator calls the `restart()` method of MergeJoin when a new pair of chunks is to be delivered. Restarting of the MergeJoin operator involves reinitializing its data structures, so that it can accept data as it is done in the procedure constructing the operator. It is effectively the same as adding GID as major key to the merge-join, but it is faster to use sandwich operators, because multi-key merge-join carries an overhead.

5.4. Relevance policy for the Cooperative Merge Join

The implementation of the Cooperative Merge Join requires modification to the Relevance functions governing scheduling of queries, loading data and evicting data (see Section 3.4).

The Cooperative Merge Join involves two CScan operators reading two tables synchronously. To implement scheduling for Cooperative Merge Join, the Relevance functions discussed in Section 3.4 need to be redefined. In the following discussion we treat Cooperative Merge Join as an object *cmj* that contains two attributes *cmj.child* for a CScan scanning the child table, and *cmj.parent* for a CScan scanning the parent table. In fact, Cooperative Merge

```
1: function CMJRelevance(cmj)
2:     if not CMJStarved(cmj) then return −∞
       return - ChunksNeeded(cmj) + WaitingTime(cmj) / RunnningCScans()
3:
4: function CMJStarved(cmj)
5:     return NumberOfAvailableChunks(cmj) < 2
6:
7: function UseRelevance(chunk, cmj)
8:     cols ← CScanColumns(cmj.child) ∪ CScanColumns(cmj.parent)
9:     U ← ‖InterestedOverlappingCScans(chunk, cols)‖
10:    P_u ← NumberCachedPages(chunk, cols)
11:    return P_u/U
12:
13: function LoadRelevance(chunk, cmj)
14:    cscan_cols ← CScanColumns(cmj.child) ∪ CScanColumns(cmj.parent)
15:    cscans ← OverlappingStarvedCScans(chunk, cscan_cols)
16:    cols ←ColumnsUsedInCScans(cscans)
17:    L ← ‖cscans‖
18:    P_l ← ‖ColumnPagesToLoad(chunk, cols)‖
19:    return L/P_l
```

Figure 5.3: Pseudo-code of the Relevance functions in the Cooperative Merge Join

Join is registered in the ABM as a pair of CScan operators that have to deliver chunks synchronously. Thus, many helper functions used for scheduling remain unchanged including `OverlappingStarvedCScans()`, `InterestedOverlappingCScans()`, `ColumnsUsedInCScans()` and `RunnningCScans()`. CScan operators that are part of a CMJ are taken into account by these functions in the same way single CScans are treated.

The implementation of ABM presented in Figure 3.4 needs to be adapted to support the Cooperative Merge Join. The main loop works in the same manner. The `ChooseCScanToProcess()` needs to iterate over both CScans and CMJs to find the most relevant one. Once a CScan or a CMJ to service is found the `ChooseChunkToLoad()` function is called to find a chunk to load. The chosen chunk and CScan or CMJ is passed to the `LoadChunk()` function which perform the actual loading of data into memory. In case a CMJ was chosen, matching chunks of both scanned tables are loaded. Evicting data and freeing memory remains unchanged.

Also, the `SelectChunk()` function requires a small adaptation. Since a CMJ is a pair of CScans that have to synchronize, the `SelectChunk()` function has to detect whether the other CScan participating in a CMJ has already selected a chunk. If one of CScans already called `SelectChunk()` and was assigned a new chunk to process, the other CScan synchronizes by choosing the matching chunk. Otherwise, if it is the first call to `SelectChunk()` after full processing of a pair of chunks, a selection based on the `UseRelevance()` function is performed.

The most basic notion used in scheduling of CScans is the number of available chunks and the number of needed chunks. They are provided by `ChunksNeeded()` and `NumberOfAvailableChunks()` functions, respectively. The same parameters can be defined for CMJs. We consider a chunk to be available if it is available for both child and parent side. Similarly, a chunk is considered needed if both sides request it.

Figure 5.3 presents Relevance functions for the Cooperative Merge Join. The `CMJRelevance()` follows the same principles as the `CscanRelevance()` function. The only difference is a call to `CMJStarved()` instead of `CScanStarved()`. The `CMJStarved()` function checks how many matching chunks are available on both sides and returns $TRUE$ if and only if the number is less than two.

The `UseRelevance()` function gathers the number of CScans that are interested in columns belonging to chunk on the child or the parent side. The less interested CScans there are, the

```
1: function AnnotateCMJ(node)
2:     if node.type = MergeJoin then
3:         (stream_size, merge_join_found) ← FindOrderedStream(node, FALSE)
4:         SetProperty(node, ORDERED_STREAM_SIZE, stream_size)
5:         if merge_join_found then
6:             SetProperty(node, OVERLAPPING_MJOIN, TRUE)
7:     for all child in node.children do
8:         AnnotateCMJ(child)
9:
10: function FindOrderedStream(node, merge_join_found)
11:     if node.type ∈ OrderDestroyingOperators then
12:         return (0, merge_join_found)
13:     else if node.type = Scan then
14:         return (Scan.num_tuples, merge_join_found)
15:     else if node.type = MergeJoin then
16:         SetProperty(node, OVERLAPPING_MJOIN, TRUE)
17:         merge_join_found ← TRUE
18:     return FindOrderedStream(node.left_child, merge_join_found)
```

Figure 5.4: Rewriter functions used to introduce the Cooperative Merge Join in query plans

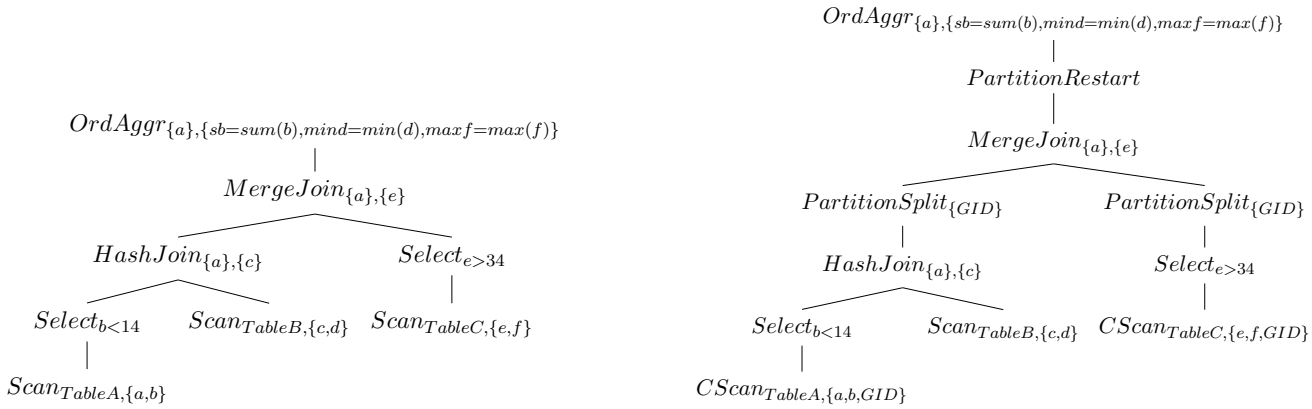
higher value of `UseRelevance()`. Similarly, `LoadRelevance()` checks the number of interested CScans on both sides and tries to find a chunk with highest ratio of interested CScans and cost of loading.

5.5. Introducing Cooperative Merge Join in query plans

Similarly to single-table Cooperative Scans, the Cooperative Merge Join is introduced in the query plan by means of appropriate functions in the Rewriter module (see Section 2.11 and 4.9). In Figure 5.4 we present the `AnnotateCMJ()` function, that annotates nodes in the query plan with two properties: $ORDERED\_STREAM\_SIZE$ and $OVERLAPPING\_MJOIN$. The $ORDERED\_STREAM\_SIZE$ indicates the size of the ordered stream arriving at certain node. If the stream is not ordered, $ORDERED\_STREAM\_SIZE$ equals 0. The $OVERLAPPING\_MJOIN$ is a boolean property that is set to $TRUE$ whenever there are two or more overlapping MergeJoin operators i.e. the output of one MergeJoin is passed to another MergeJoin as input without changing ordering of the stream. Such scenario is not supported by the ABM, as it only handles 2-way Cooperative Merge Join. The default values of $ORDERED\_STREAM\_SIZE$ and $OVERLAPPING\_MJOIN$ properties are 0 and $FALSE$ respectively.

The `AnnotateCMJ()` function traverses the tree in top-down fashion and detects all MergeJoin nodes. For each of them, a helper function `FindOrderedStream()` is executed. Its purpose is to determine the value of $ORDERED\_STREAM\_SIZE$ and set $OVERLAPPING\_MJOIN$ if needed. It returns two results: the size of the stream that has been found and a flag $merge\_join\_found$ that indicates if there is a overlapping MergeJoin below. To determine the size of ordered stream, the `FindOrderedStream()` goes down the tree following a path that an ordered stream is generated. First of all, it stops on any of operators that change ordering of the stream by inspecting whether the type of the node belong to the *OrderDestroyingOperators* set. This set includes: Sort, TopN and Aggregation. None of join operators are considered to destroy the order of incoming stream. Indeed, both MergeJoin and HashJoin preserve the order of tuples arriving at the left side. Thus, the `FindOrderedStream()` function takes the left-most path possible. When it encounters a Scan operator the size of the stream can be finally returned.

After annotating the tree with the $ORDERED\_STREAM\_SIZE$ and



(a) Original merge-join query plan

(b) Query plan using Cooperative Merge Join as introduced in Section 5.5

Figure 5.5: An example of introduction of the Cooperative Merge Join in a query plan

$OVERLAPPING\_MJOIN$ it is possible to introduce the Cooperative Merge Join. Two conditions need to be met to do so: there must be no overlapping MergeJoins, as this situation is not supported and the size of the stream has be larger than specified threshold. We omit the procedure that transforms the query plan to use the Cooperative Merge Join for brevity. Figure 5.5 shows an example query plan transformation. As described in Section 5.3 the MergeJoin operator is „sandwiched" between PartitionSplit and PartitionRestart. Moreover, both Scan operators that provide ordered data streams for the MergeJoin operator are turned into CScan operators. Both of them are assigned the same identifier, so that the Active Buffer Manager will be able to detect that they are part of the same Cooperative MergeJoin and need to be synchronized. Note, that the third Scan operator that feeds the right side of HashJoin is not changed into a CScan. In fact, it can also be modified, as the right side of HashJoin accepts data delivered out-of-order. This case is handled by the rules for introducing Cooperative Scans described in Section 4.9. Thus, applying both rules for Cooperative Scans and Cooperative Merge Join results in a query plan fully exploiting the potential of the Cooperative Scans framework.

5.6. Handling of checkpoints

In Section 4.7 we present how metadata in the ABM is influenced by the process of checkpointing introduced in Section 2.9. So far, we focused on checkpointing of a single table. In case two or more tables are clustered by means of join indices (see Section 2.6) the checkpointing mechanism needs to rebuild all tables that depend on the checkpointed table. Figure 7.2 depicts an example cluster tree (see Section 2.6.1). Let us suppose the nation table is to be checkpointed. Modifying data in the table that other tables are clustered on implies necessity of rebuilding the referencing tables, as the order of tuples in the referenced table may change. Thus, checkpointing of the nation table triggers checkpointing of the customer and supplier tables. Also, a checkpoint of the parent table is triggered by the checkpoint of child table. In the discussed scenario, checkpointing of the nation table triggers checkpoint of the region table. This behaviour is caused by the fact that the Join Index Summary (see Section 2.6.2) structures need to be rebuild, as they depend on both parent and child table. Consequently, a checkpoint of a table that belongs to a cluster tree triggers checkpointing of the whole cluster

tree.

After a checkpoint of the cluster tree is finished, new versions of each table are created in the master catalog. Since the proper functioning of Cooperative Merge Join depends on correct partitioning of tables (see Section 5.2), the partitioning has to be rebuild. Chunk boundaries have to be aligned to the same key boundaries on both the parent and the child side. When a new (checkpointed) version of the table is detected in the ABM, the partitioning for all tables in the same cluster tree has to be done again. In principle, the ABM has to perform an analogous algorithm that is done in the checkpointing process i.e. find all tables belonging to the same cluster tree and perform a specified operation for them.

5.7. Summary

This chapter presented a solution to the problem of support for operators requiring an ordered data stream. We introduced the Cooperative Merge Join, a novel algorithm that extends Cooperative Scans with an ability to support join processing. Cooperative Scans and Cooperative Merge Join constitute a fully-functional buffer management framework that supports all operators used in VectorWise.

This chapter finishes the discussion about design and implementation of Cooperative Scans. The following chapters evaluate the performance of both Cooperative Scans with Cooperative Merge Join and the Predictive Buffer Manager solutions with benchmarks. Chapter 6 presents results of custom-designed benchmarks, whereas Chapter 7 provides evaluation with the TPC-H benchmark.

Chapter 6

Synthetic benchmarks

In this chapter we present performance results obtained in synthetic benchmarks designed to evaluate the improvements achieved by Cooperative Scans described in Chapters 3, 4 and 5, as well as the Predictive Buffer Manager (PBM) described in Section 4.3.

6.1. The basic benchmark

To evaluate the performance of Cooperative Scans and the Predictive Buffer Manager we tested the system in a simple scenario, which is based on the TPC-H benchmark [Tra11]. TPC-H is an industry-standard benchmark for ad-hoc decision support systems. It consists of a schema containing 8 tables and 22 parametrized queries. A more detailed analysis of the TPC-H benchmark can be found in Section 7.1. We chose two out of 22 available queries that do not perform any join operations, thus scan a single table. The first TPC-H query (Q1) scans the lineitem table performing relatively CPU intensive computations. The TPC-H query 6 (Q6) scans the same table performing a simple aggregation and consequently is less CPU-intensive than Q1. We denote TPC-H Q1 by S (slow query) and TPC-H Q6 by F (fast query). Each query is parametrized by the percentage of the whole table that it scans e.g. we denote TPC-H Q1 scanning 50% of the table by S-50. The beginning of the range is chosen randomly. We used ranges of 1%, 10%, 50% and 100% of the table.

To simulate a concurrent workload we run queries in 16 parallel streams, each consisting of 4 randomly chosen queries ran sequentially. For each query type we gather the average execution time, as well as its execution time in a system with no other queries running in parallel and empty buffer pool (basetime). This allows us to calculate the normalized query latency defined as the average query execution time in the benchmark run divided by its basetime. Also, we provide system-wide results such as the total execution time, average stream time, average normalized query latency, the total volume of I/O performed and average CPU utilization.

We used a TPC-H scale factor 40 dataset, where the lineitem table contains slightly over 240 million tuples. In this experiment the total volume of data accessed by the queries amounts to 2 GB. We use a buffer pool of 800 MB, hence capable of caching 40% of the data. In all benchmarks presented in this chapter we use a chunk size of 1 million tuples, so the lineitem table is divided into 240 chunks.

All benchmarks presented in this chapter were performed on a server equipped with two quad-core Intel(R) Xeon(R) X5560 CPUs and 48 GB of RAM memory. The used storage was a 2-way RAID-0 matrix delivering up to 140 MB/s of bandwidth. Hence, we expect the system to be mostly I/O bound, so improvements with respect to the buffer management

	LRU			PBM			CScans		
System statistics									
avg stream time [s]	149.57			98.21			44.71		
avg. norm. latency	6.58			5.51			2.40		
total time [s]	200.03			126.49			65.64		
CPU use [%]	65.80			90.30			159.00		
total I/O [MB]	20461.61			11694.86			5660.02		
Query statistics									
query	avg. [s]	std dev.	norm. lat.	avg. [s]	std dev.	norm. lat.	avg. [s]	std dev.	norm. lat.
F-1	1.92	87.63	18.73	1.52	49.46	14.83	0.77	22.09	7.50
F-10	9.92	74.74	5.48	12.05	35.38	6.65	3.06	18.58	1.69
F-50	39.76	11.59	4.95	28.42	12.19	3.54	11.48	3.59	1.43
F-100	84.55	1.33	5.29	46.53	5.40	2.91	21.17	3.48	1.32
S-1	2.63	66.96	5.76	2.83	40.27	6.21	0.75	27.22	1.65
S-10	11.73	74.02	4.65	10.60	41.30	4.20	5.16	29.18	2.04
S-50	44.66	3.38	4.31	36.21	10.43	3.49	21.99	9.86	2.12
S-100	70.39	21.34	3.50	44.62	6.69	2.22	28.42	8.05	1.41

Table 6.1: The basic experiment with a set of FAST and SLOW queries scanning 1%, 10%, 50% and 100% of the lineitem table

should influence the performance perceived by the user.

The results are presented in Table 6.1. We can observe Cooperative Scans considerably outperforming the LRU and the Predictive Buffer Manager in all system-wide results. There is a fourfold improvement in the amount of performed I/O in comparison to the LRU buffer manager. It translates into a roughly threefold advantage in other system-wide results. At the same time, thanks to increased buffer reuse Cooperative Scans achieve higher CPU utilization. However, the system still remains I/O bound, as with CPU utilization of 159% it uses on average merely 2 out of 8 CPU cores it is equipped with.

Also, the Predictive Buffer Manager provides an improvement over the LRU buffer manager in all results. It decreases the amount of performed I/O by 43%, which translates into a 34% decrease of the average stream time and 37% of the total time. The average normalized latency is improved by 16%.

By taking a closer look at individual query results we can explain the lower improvement in average normalized query latency. The PBM mostly improves the normalized latency of long queries (F-100, S-100, F-50 and S-50). The normalized latency of short queries (F-10, S-10, F-1 and S-1) is decreased slightly or increased in some cases.

On contrary, Cooperative Scans improve the normalized query latency of all used query types. As a result, the average normalized latency is much lower for Cooperative Scans. This phenomenon is explained by the fact that Cooperative Scans use a better scheduling of queries using the `CscanRelevance()` function (see Figure 3.3) that prioritizes shorter queries. Besides lower I/O consumption it provides an additional advantage of Cooperative Scans in a highly concurrent workload with queries of varied length.

6.2. Sharing potential analysis

To understand the performance results better, we provide an analysis of the sharing potential. In a system loaded with multiple concurrently working queries we can calculate the volume of data that is needed at some moment by only 1 scan, exactly 2 scans etc. The result of

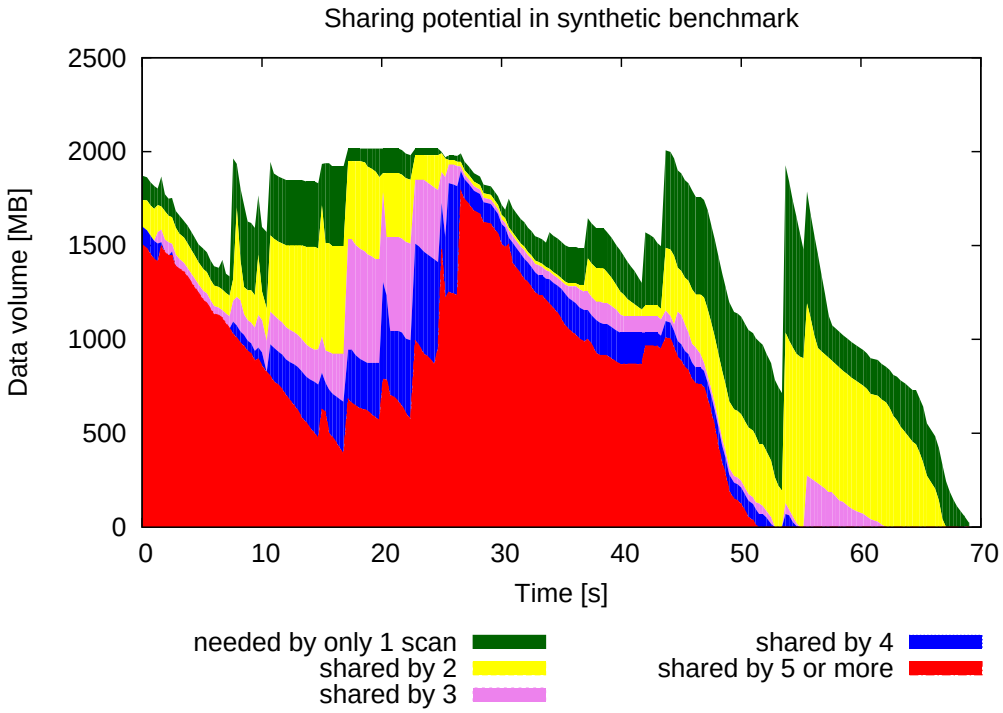


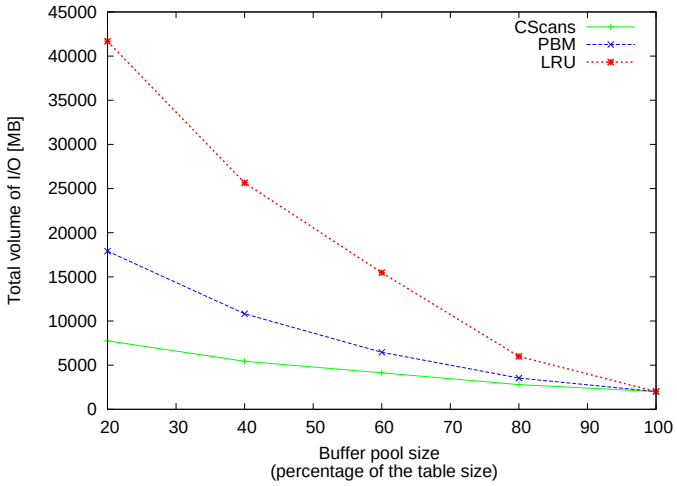
Figure 6.1: Sharing potential in the synthetic benchmark

such an analysis is depicted in Figure 6.1. The green area indicates the amount of data that is requested by a single scan, thus is not subject to be reused by any other query. The area marked with red represents the data that is needed by 5 or more scans working in the system at certain moment. In Figure 6.1 the red component dominates, showing that a high volume of data can be reused by a large number of queries. Consequently, it explains the strong advantage of Cooperative Scans in the presented benchmark. The sharing potential is also exploited by the Predictive Buffer Manager to some extent. However, since the data has to be delivered in-order it is impossible to attain a level of buffer reuse on par with Cooperative Scans. The data that has to be loaded in specified order results in eviction of already cached pages, regardless of the fact that they may be accessed in the near future.

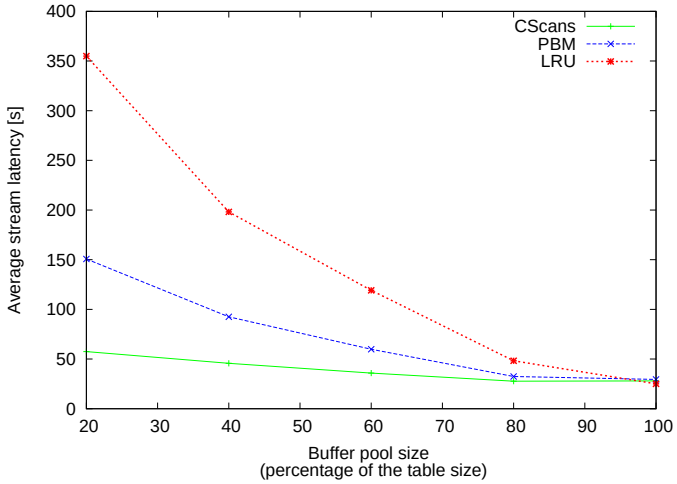
6.3. Scaling data volume

The size of dataset that is scanned is an important factor influencing performance in I/O bound systems. To simulate a growing dataset we perform experiments with varying size of the buffer pool size ranging from 20% to 100% of the total amount of scanned data. We use the same parameters as in the benchmark found in Section 6.1. In Figure 6.2 we present three main system-wide results: total volume of performed I/O (a), average stream time (b) and average normalized latency (c) in relation to the buffer pool capacity.

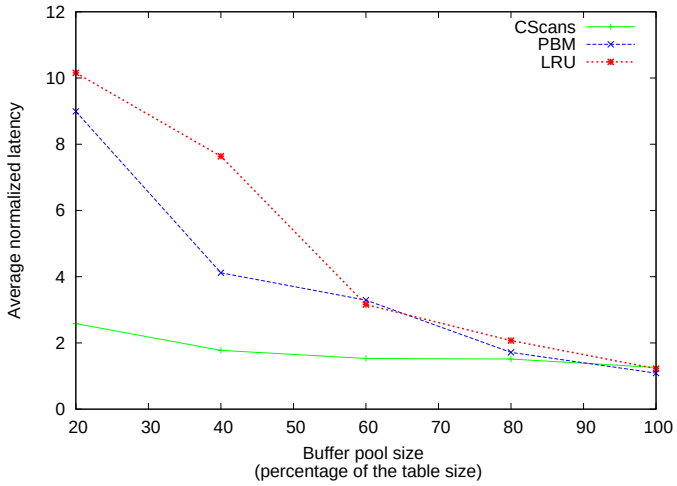
First off all, we observe superior performance of Cooperative Scans in all the presented results for all sizes of the buffer pool. The performance of Cooperative Scans appears to be insensitive to the capacity of the buffer. The flexibility of out-of-order delivery allows to save a considerable amount of I/O even in the case of a small buffer pool. Only a slight difference in the volume of I/O performed on the buffer pool caching 20% of the table and the buffer pool caching all data suggests, that the streams are short enough to be serviced by a loading



(a) Total volume of I/O



(b) Average stream time



(c) Average normalized latency

Figure 6.2: Performance of buffer management solutions under varying size of the buffer pool

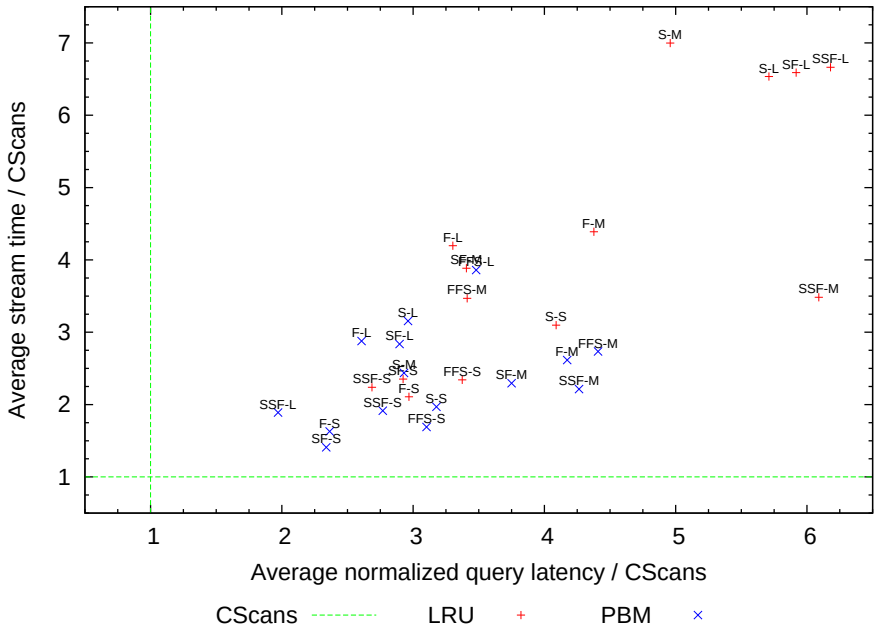


Figure 6.3: Performance of various buffer management solutions under varying query sets and scanned ranges

most of the data only once.

The Predictive Buffer Manager exhibits a considerable improvement over the LRU buffer manager, but is subject to performance degradation under small buffer pool as well. The results of average normalized latency differ from two other plots. In comparison to the LRU buffer manager, the PBM improves average normalized latency only for the buffer pool holding less than 40% of the relation. As mentioned before, the PBM uses the same way of scheduling I/O requests as the LRU buffer manager, which does not prioritize shorter queries. In some cases, the decreased need for I/O achieved by the PBM results in lowered average normalized latency, but there is no general rule.

6.4. Exploring query mixes

In Section 6.1 we prove the superior performance of Cooperative Scans for a predefined set of queries and possible ranges they access. To verify and analyze the performance benefits, we perform experiments with varying query sets and ranges. We provide results of two parameters: average stream and average normalized latency as depicted in a two-dimensional plot in Figure 6.3. The point (1,1) represents the result of an experiment with Cooperative Scans enabled. Other points are plotted according to the relative performance of both LRU and Predictive Buffer Manager in comparison with the run with Cooperative Scans. The labels indicate query mix and used ranges in format „<SPEED>-<SIZE>". <SPEED> indicates the number and type of queries included e.g. FS - mix of F and S queries, F - only F ones, FFS - 2 times more F queries than S ones etc. <SIZE> represent a set of fractions that are used to generate ranges: S – short (mix of queries reading 1, 2, 5, 10 and 20% of a table), M - mixed (1,2,10,50,100) and L - long (10,30,50,100).

The plot depicted in Figure 6.3 proves the advantage of Cooperative Scans in all cases. However, the relative improvement depends on query set. In comparison to the LRU buffer

manager, Cooperative Scans have much stronger advantage in the case of long queries. Indeed, the LRU buffer manager is subject to low buffer reuse under workload with long scans. In particular, LRU is the worst possible policy for a sequence of full-table scans. In such scenario the buffer always keeps a „tail" of a table, whose length depends on the size of the buffer pool. When a next scan is started, the data can be reused only if the whole table fits the memory. Otherwise, the whole stored tail is evicted resulting in no data reuse. For short queries chances that recently used data will be reused increase, thus the LRU policy performs better getting close to the Predictive Buffer Manager in some cases.

The PBM performs relatively well for all types of queries. It also exhibits performance comparable to Cooperative Scans for short queries, but there is stronger degradation when longer queries are involved.

6.5. Scaling the number of concurrent queries

As Cooperative Scans are designed to improve performance in concurrent workload, their advantage should be higher with increasing number of concurrent streams. In the following experiment we use the F query introduced in Section 6.1 . We ran two variants, one with short queries scanning 10% of the relation and second with queries scanning half of the relation. Each stream consists of 4 queries, the starting position of scan is chosen randomly for each of them. The experiment is performed for the number of streams ranging from 1 to 32. The results can be found in Figure 6.4.

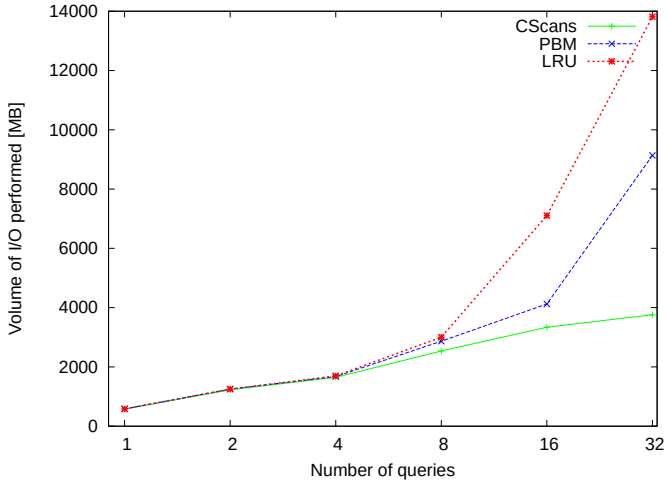
As the number of concurrent streams increases, the advantage of Cooperative Scans becomes more visible. In particular, in the case of queries scanning 50% of the relation the performance remains virtually constant. With many concurrently working long queries scanning exactly the same set of columns, we expect a large fraction of queries to be satisfied with each of loaded chunks. Thus, as long as the system is not CPU-bound the performance is insensitive to the number of parallel streams.

The Predictive Buffer Manager is capable of saving a large amount of I/O in comparison with the LRU buffer manager, but it does not exhibit the same flat characteristic of Cooperative Scans. As the data has to be delivered in-order queries can not „attach" to the data that is being loaded. On contrary, many queries scanning at different positions in the table start fighting for I/O. The PBM performs better with shorter queries than with longer queries. In the case of queries scanning 10% of the relation its performance is close to the performance of Cooperative Scans for up to 16 streams, whereas for long queries it performs similarly for up to 8 streams.

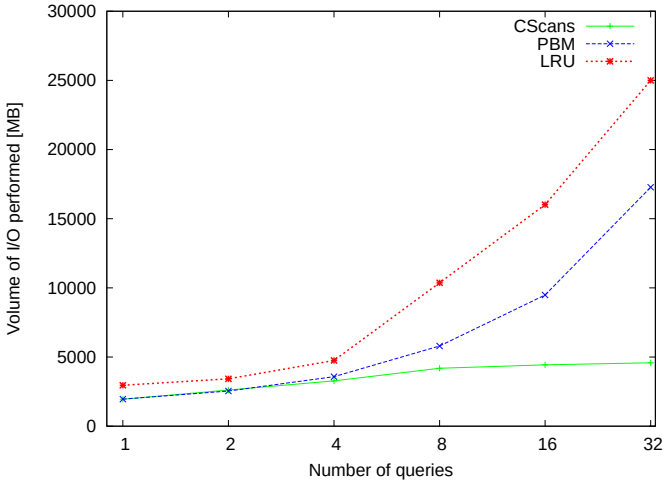
6.6. Investigating the performance of scans over multiple columns

The sharing potential in DSM storage depends on columns that are scanned by the queries (see Section 2.4). The more scanned columns are shared, the more advantage we can expect from using Cooperative Scans. To evaluate the performance of queries scanning different sets of columns we run a synthetic benchmarks on a database containing a single table with 6 attributes (from A to F). The table contains 200 million tuples, each attribute is 8 bytes wide, resulting in the total data volume of 9 GB. We use a buffer pool that is capable of caching 40% of the relation. The queries scan a random 40% of the relation and columns as specified in Table 6.2. They are issued in 16 parallel streams of 4 queries.

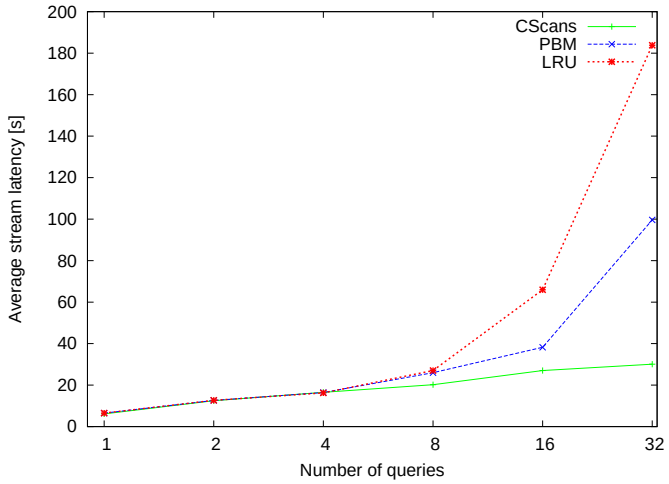
The first experiment with a single query scanning columns ABC confirms the advantage of Cooperative Scans over the two variants of the buffer manager. However, the relative



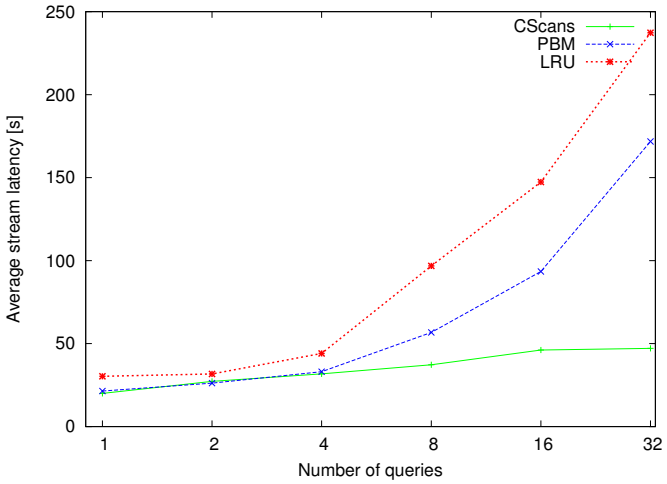
(a) Total volume of I/O, 10% queries



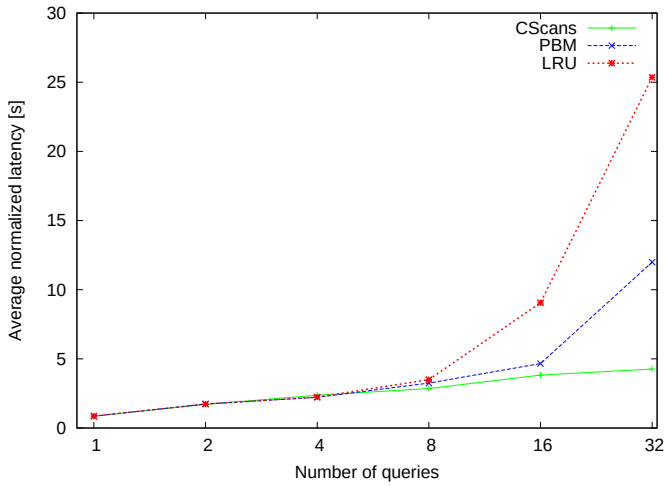
(b) Total volume of I/O, 50% queries



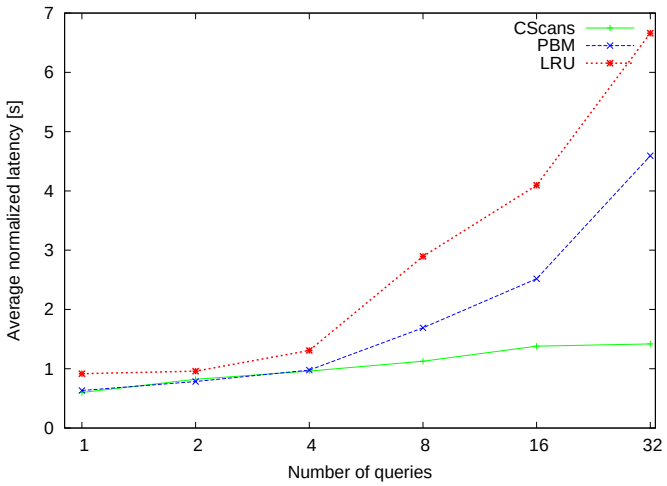
(c) Average stream time, 10% queries



(d) Average stream time, 50% queries



(e) Average normalized latency, 10% queries



(f) Average normalized latency, 50% queries

Figure 6.4: Performance results with varying number of concurrent streams

Queries (used columns)	LRU			PBM			CScans		
	total I/O [MB]	stream avg. [s]	norm. avg. latency	total I/O [MB]	stream avg. [s]	norm. avg. latency	total I/O [MB]	stream avg. [s]	norm. avg. latency
ABC	9255	62.92	0.72	6478	59.74	0.64	5679	46.91	0.53
ABC,DEF	45850	472.54	5.23	27846	264.54	2.95	19134	148.78	1.75
ABC,BCD	25264	255.84	3.06	13620	135.58	1.47	11013	103.21	1.26
ABC,BCD,CDE	27300	285.20	3.32	18293	184.02	2.19	14909	133.23	1.56
ABC,BCD,CDE,DEF	42462	475.44	5.55	27409	300.49	3.37	22718	167.64	1.99

Table 6.2: Performance of DSM queries when scanning different sets of columns of a synthetic table

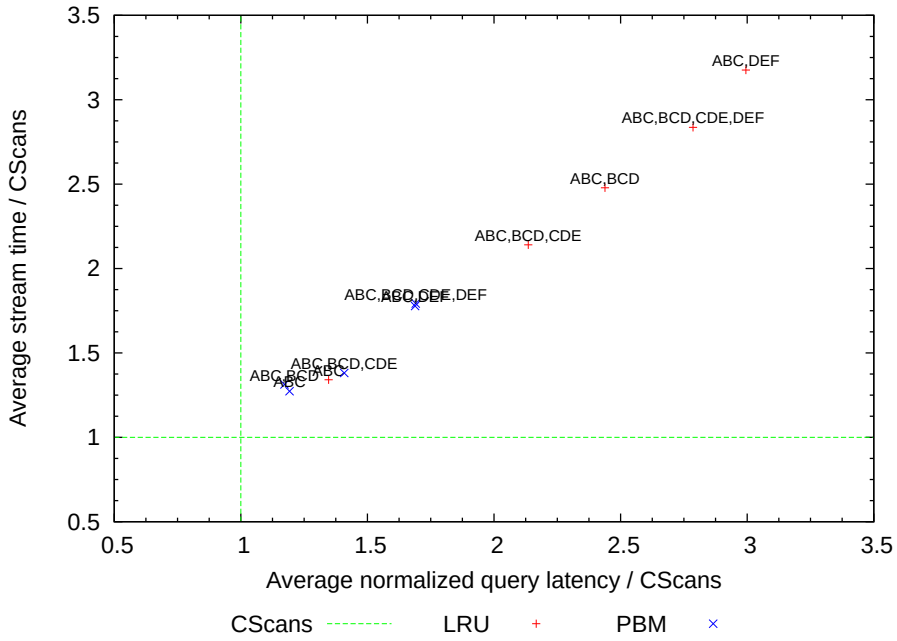


Figure 6.5: Performance of various buffer management solutions with queries scanning varying sets of columns. Results from Table 6.2, in comparison to Cooperative Scans.

advantage is lower than in the previous experiments, as with only three columns scanneda the data volume is lower, thus buffer reuse is higher for all solutions. In particular, the average normalized latency drops below 1 due to very high buffer reuse (queries run faster than in a system with empty buffer pool). After adding an additional query type scanning next three columns (D, E and F) the performance drops for all policies, but the relative improvement of Cooperative Scans increases considerably.

Additional experiments with increasing number of overlapping sets of columns show consistent advantage of Cooperative Scans. The relative improvement is highest for the last presented query set accessing all columns from A to F with queries scanning different overlapping sets of columns.

Figure 6.5 depicts the scatter plot showing the average stream time and average normalized latency of the LRU and the Predictive Buffer Manager in relation to Cooperative Scans. The plot confirms the advantage of Cooperative Scans and the PBM. The only case where LRU buffer manager is close to them is the simplest set with queries scanning columns A, B and

```
SELECT
  l_shipmode,
  SUM(o_totalprice),
  SUM(CASE
    WHEN o_orderpriority = '1-URGENT'
      OR o_orderpriority = '2-HIGH'
        THEN 1
      ELSE 0
    END) AS high_line_count,
  SUM(case
    WHEN o_orderpriority <> '1-URGENT'
      AND o_orderpriority <> '2-HIGH'
        THEN 1
      ELSE 0
    END) AS low_line_count
FROM
  orders,
  lineitem
WHERE
  o_orderkey = l_orderkey
  AND l_shipmode in ('MAIL', 'SHIP')
  AND MONTH(l_commitdate) = MONTH(l_receiptdate)
  AND MONTH(l_shipdate) = MONTH(l_commitdate)
GROUP BY
  l_shipmode
ORDER BY
  l_shipmode;
```

(a) Query CMJ1

```
SELECT
  o_orderpriority,
  SUM(o_totalprice),
  COUNT(*) AS order_count
FROM
  orders
WHERE
  MONTH(o_orderdate) >= 2
  AND EXISTS (
    SELECT
      *
    FROM
      lineitem
    WHERE
      l_orderkey = o_orderkey
      AND l_commitdate < l_receiptdate
  )
GROUP BY
  o_orderpriority
ORDER BY
  o_orderpriority;
```

(b) Query CMJ2

```
SELECT
  SUM(l_extendedprice) AS revenue
FROM
  lineitem
WHERE
  l_shipdate >= DATE '1994-01-01'
  and l_commitdate < DATE '1994-01-01'
  and l_receiptdate > DATE '1987-07-03';
```

(c) Query L1

```
SELECT
  SUM(o_totalprice),
  SUM(CASE
    WHEN o_orderpriority = '1-URGENT'
      OR o_orderpriority = '2-HIGH'
      THEN 1
    ELSE 0
  END) AS high_line_count
FROM
  orders
WHERE
  o_orderdate > DATE '1997-09-23'
```

(d) Query O1

Figure 6.6: Queries used in the benchmark evaluating the performance of the Cooperative Merge Joins

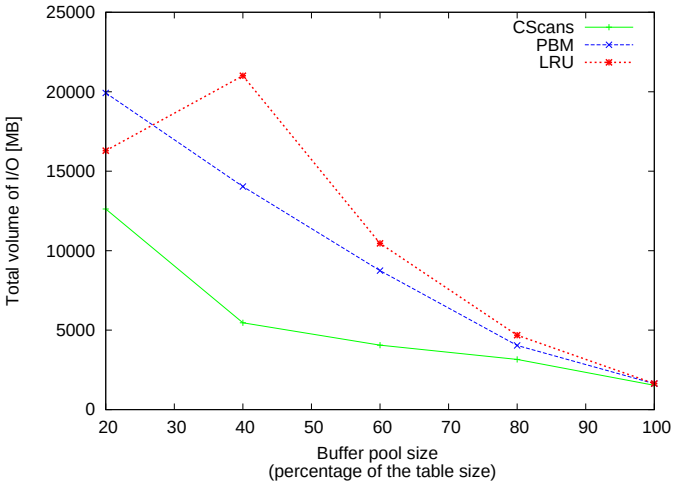
C, which as discussed before, exhibits highest buffer reuse.

6.7. Cooperative Merge Join experiments

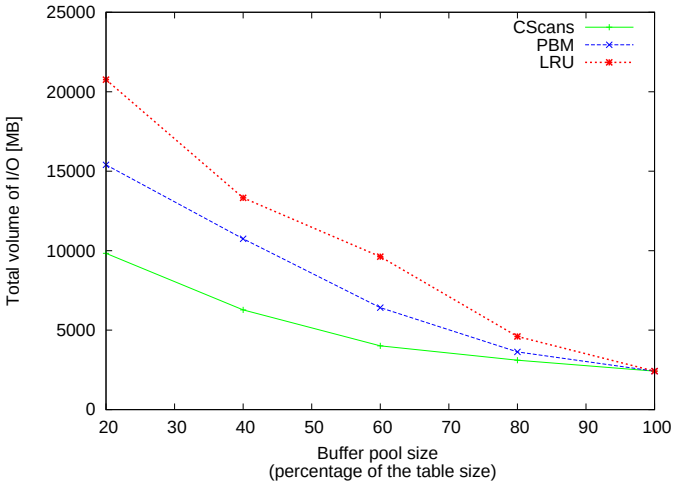
To evaluate the performance of Cooperative Merge Join we propose a set of synthetic benchmarks resembling the experiments discussed in Section 6.1. We use two largest tables of the TPC-H benchmark [Tra11]: lineitem and orders. The lineitem table references orders by means of a foreign key. The VectorWise DBMS can exploit clustering (see Section 2.6) and store lineitem sorted according to the order of the `orderkey` column of the orders table. Consequently, the two tables can be joined using the MergeJoin operator, as well as use Cooperative Merge Join (CMJ).

The TPC-H benchmarks contains two queries that perform a join of lineitem and orders without accessing other tables. We use them to as a base to create queries for the synthetic benchmark. Used queries are presented in Figure 6.6. The standard TPC-H Q4 and Q12 queries were modified to scan more columns, thus increasing the need for I/O. To check the performance of a system running both queries using the CScan operator and Cooperative Merge Join, we provide two simple queries scanning lineitem and orders respectively.

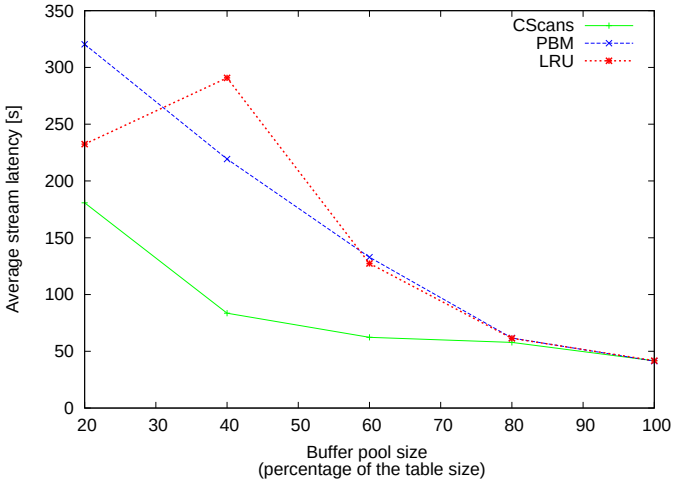
We ran two variants of the experiment: one comprised of CMJ1 and CMJ2 queries and a second one additionally using L1 and O1 queries accessing single table (CScan queries).



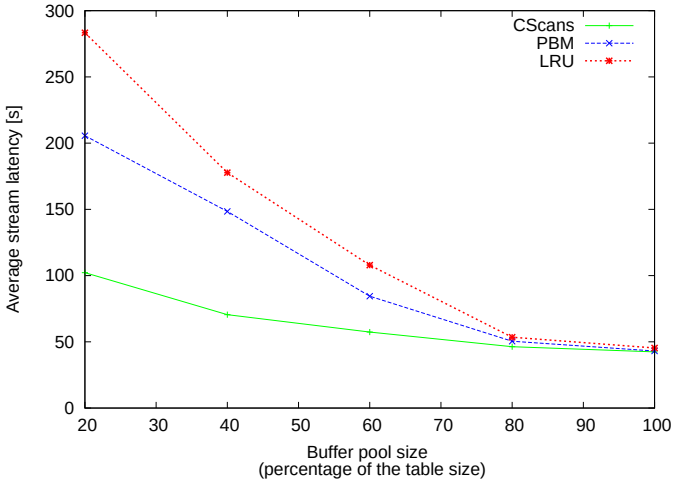
(a) Total volume of I/O, CMJ queries



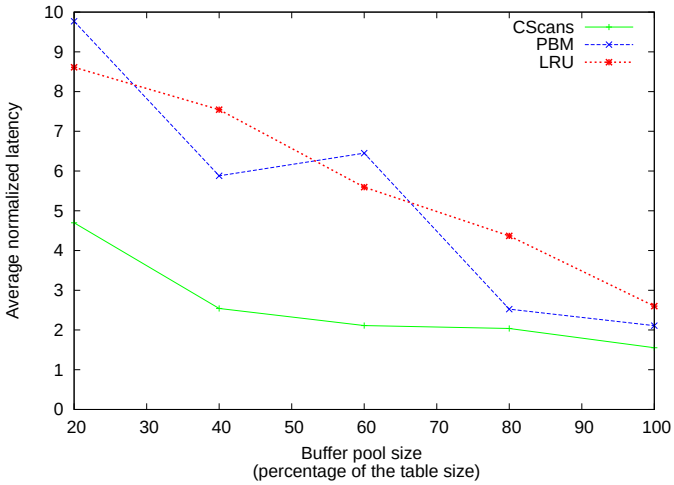
(b) Total volume of I/O, CMJ and CScan queries



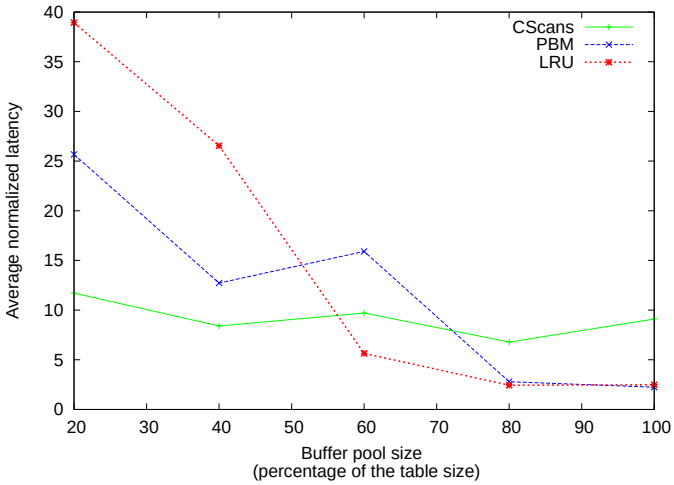
(c) Average stream time, CMJ queries



(d) Average stream time, CMJ and CScan queries



(e) Average normalized latency, CMJ queries



(f) Average normalized latency, CMJ and CScan queries

Figure 6.7: Performance of various buffer management solutions under varying size of the buffer pool in a test using merge-join queries

The first query mix is intended to check performance of Cooperative Merge Join in isolation, whereas the second variant verifies how CMJ queries work mixed with single-table CScans. As in the previous synthetic benchmarks each query scans a random fraction of a table (1%, 10%, 50% or 100%) starting at a random position. For CMJ1 and CMJ2 queries the range is applied to the child table (lineitem) and propagated to the parent table using the Join Index Summary (JIS) structure (see Section 2.6.2). The JIS structure contains 917 evenly distributed entries, thus the range propagation mechanism provides accurate estimate of the matching range on the parent side. We focus on experiments with varying size of the buffer pool. We check buffer pools storing from 20% up too 100% of the whole needed data volume. Note that the amount of accessed data depends on used queries and scanned columns. In the experiment with CMJ queries the volume amounts to 1.5 GB, whereas for in the run with a mix of CMJ and CScan queries it increases to 2.5 GB, as CScan queries scan columns that are not used by CMJ queries.

The results are depicted in Figure 6.7. In general, the same pattern as in Figure 6.2 can be observed. The system using Cooperative Scans and Cooperative Merge Join shows a clear advantage. However, we no longer observe a nearly flat curve of Cooperative Scans. The lower relative improvement provided by Cooperative Scans working on a small buffer pool is a result of increased number of accessed columns and variety of queries, thus lower sharing opportunities.

Figure 6.7 (f) shows the only graph where the advantage of Cooperative Scans cannot be observed for all tested sizes of the buffer pool. To explain that we need to take a closer look at individual normalized query latencies. It turns out that with Cooperative Scans the O1-1 query, which is the shortest query in the experiment is delayed much more. Its normalized latency ranges from 60 to 100 with Cooperative Scans, whereas it equals to about 3 for both the LRU buffer manager and the Predictive Buffer Manager. The large difference contributes strongly to the average normalized latency of the whole experiment. This phenomenon seems to be effect of loading data in chunk-at-time manner. With this approach very short queries have to wait until large chunks that were scheduled earlier are loaded. It happens regardless of the early scheduling of short queries with the `CScanRelevance()` function. A query that enters the system, even when scheduled early, has to wait until already scheduled chunks are loaded. Contrary to the LRU and PBM solutions, there is no mechanism to „cut the line" by issuing a blocking I/O requests that is carried out by the buffer manager as soon as possible. This suggests that for very short queries dropping Cooperative Scans may be needed.

In the case of small buffer pool size Cooperative Scans maintain the advantage in average normalized latency. Similarly to other experiments, the PBM exhibits results similar to the old buffer manager due to lack of dedicated query scheduling mechanism in both of them.

6.8. Summary

This chapter presented a variety of synthetic benchmarks designed to evaluate the performance of Cooperative Scans along with Cooperative Merge Join, the Predictive Buffer Manager and the LRU buffer manager. The results confirm that the solutions proposed in this thesis have a potential to provide high performance improvements. However, we state that the perceived performance improvement strongly depends on used environment and benchmark. The next chapter provides evaluation of performance in the TPC-H throughput benchmark.

Chapter 7

Performance evaluation with TPC-H

This chapter presents performance results obtained in the TPC-H throughput benchmark. We evaluate the LRU buffer manager (see Section 4.2), Cooperative Scans along with the Cooperative Merge Join (see Chapters 3, 4 and 5), as well as the Predictive Buffer Manager (PBM) (see Section 4.3).

7.1. The TPC-H benchmark

TPC-H [Tra11] is an industry standard benchmark for evaluating large scale decision support applications. TPC-H schema consists of eight tables with 61 columns in total. The size of the database can be changed using the scale factor parameter (SF). The default value is 1 which corresponds to 1 GB of uncompressed data. TPC-H contains 22 queries which are characterized by high degree of complexity.

The TPC-H benchmark is divided into two main parts. The TPC-H „power" run is aimed at evaluating performance of queries executed in isolation i.e. with no other operations in parallel. On the other hand, the purpose of the TPC-H „throughput" run is to test the performance of concurrent workload. The TPC-H throughput benchmark involves running several parallel streams of queries. The TPC-H benchmark specifies minimum number of parallel streams depending on scale factor. In experiments presented in this chapter we use 10 parallel streams. Each of them executes all of 22 TPC-H queries in random order and with random parameters. To assure reproducibility of results the whole benchmark run is parametrized with a seed value $seed_{benchmark}$. Query parameters and their order in each stream are created with a pseudo-random number generator that is passed the initial seed value $seed_{stream}$. The value of $seed_{stream}$ of the i-th stream equals $seed_{benchmark} + i$. Thus, running the benchmark with the same seed value assures the same queries to be executed.

Keep in mind these are in no way official TPC-H results, we are only using queries and the schema as an example. The full benchmark also has update queries, extra functionality tests etc. Here we only present the TPC-H throughput run without updates i.e. the contents of the database do not change during the experiment.

7.1.1. TPC-H schema and cluster trees

Figure 7.1 presents the TPC-H schema. The database consists of 8 tables, which have 61 attributes in total. The number of tuples in most tables depends on the Scale Factor (SF) and is determined as presented in Figure 7.1. The two largest tables in terms of both the number of tuples and the number of columns are „lineitem" and „orders". The arrows in

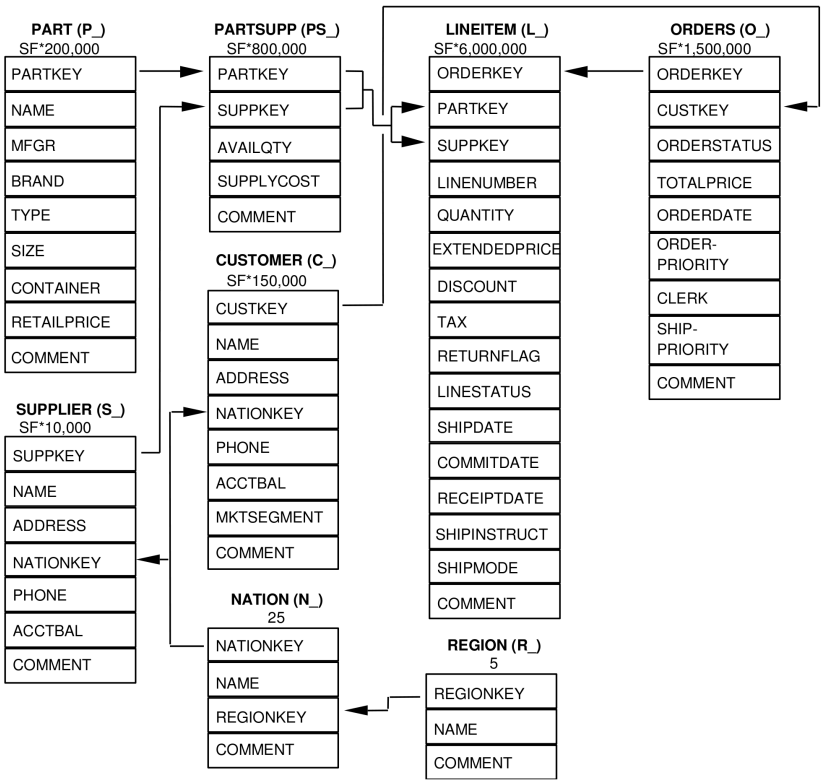


Figure 7.1: The TPC-H schema (from [Tra11])

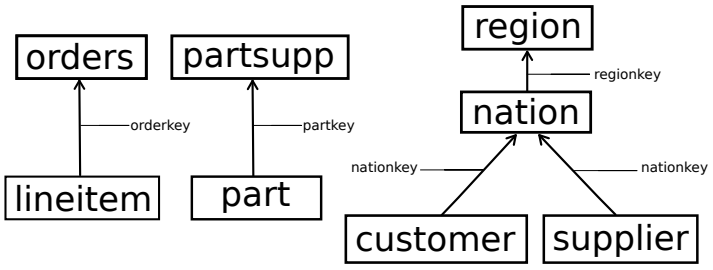


Figure 7.2: Cluster trees we propose in the TPC-H benchmark

Figure 7.1 show foreign key relationships. As described in Section 2.6 and 2.6.1 VectorWise can cluster tables that are related by means of a foreign key. Figure 7.2 depicts a cluster tree that we propose for the TPC-H schema. As we can see, the largest table lineitem is clustered on orders. Besides, two other large tables part and partsupp are clustered. The last cluster forms a tree with region defining the order of tuples in nation, which influences ordering of supplier and customer.

7.1.2. Static sharing potential analysis

Before presenting benchmark results we first analyze the theoretical improvement that can be attained with Cooperative Scans and the Cooperative Merge Join. From our perspective the most important aspect is I/O volume read by each query.

Table 7.1 presents how much data is shared by each pair of queries in a single stream with the seed value of 411193435. The dataset used was TPC-H scale factor 40, which requires

Q1	**1852**																					
Q2	0	**272**																				
Q3	64	0	**133**																			
Q4	0	0	16	**39**																		
Q5	158	2	41	1	**498**																	
Q6	299	0	27	0	158	**299**																
Q7	443	1	102	18	169	65	**1214**															
Q8	317	15	57	19	9	2	977	**1002**														
Q9	1267	224	79	19	459	257	1055	921	**3740**													
Q10	47	0	5	0	73	41	2	2	61	**959**												
Q11	0	158	0	0	1	0	1	1	158	0	**213**											
Q12	45	0	15	3	70	42	36	1	99	18	0	**271**										
Q13	0	0	18	0	36	0	83	72	0	15	0	0	**3407**									
Q14	81	12	0	0	162	81	73	12	167	1	0	14	0	**189**								
Q15	111	15	0	0	0	0	0	0	223	0	0	0	0	0	**270**							
Q16	0	180	0	0	0	0	0	8	153	0	91	0	0	8	0	**198**						
Q17	1145	4	46	0	363	232	847	732	2617	36	0	0	0	159	211	5	**2628**					
Q18	229	0	63	19	112	46	247	222	719	166	0	99	352	0	0	0	229	**2475**				
Q19	1267	10	53	0	382	257	890	769	2739	41	0	26	0	167	223	11	2628	229	**3000**			
Q20	90	99	0	0	1	0	343	304	733	0	147	0	0	4	2	91	350	47	350	**833**		
Q21	0	1	24	35	295	0	238	141	1015	18	1	193	0	95	131	0	772	483	**772**	1	3510	
Q22	0	0	18	0	36	0	83	72	0	145	0	0	234	0	0	0	0	234	0	0	0	**364**
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22

Table 7.1: The volume of data in MB shared by each pair of TPC-H queries on scale factor 40 database. The top line represents the volume of data accessed by query stand-alone.

10 GB to be stored in compressed column-oriented storage of VectorWise. The last column of each row indicates how much data is needed by each of queries by itself. We can see that the most demanding queries in terms of I/O are Q09, Q20, Q19, Q17 and Q18 that need over 2 GB of data. First of all, we observe many query pairs that share no or very little data. Such queries must scan either different tables, columns or ranges, so that there is no overlap. Interestingly, query Q21 that scans 3.5 GB of data, yet does not share more than 1 GB with any of other queries. The pairs that share most are (Q9, Q19), (Q17, Q19) and (Q9, Q17) that share 2739 MB, 2626 MB and 2617 MB respectively. So, in fact Q09, Q17 and Q19 form a cluster that accesses nearly the same data. These are the only query pairs that share more than 2.5 GB of data. The other query pair share less than 1.5 GB.

In general, we conclude that the amount of data needed by each of the queries is relatively low in comparison to the size of the whole dataset. Indeed, 14 out of 22 queries access less that 1 GB, which is 10% of the database size. Moreover, most of query pairs share little or no data. Note that with multiple streams the sharing potential between queries belonging to different streams with different seed value may differ.

7.2. Performance results

7.2.1. Test environment

We ran the TPC-H throughput benchmark with the value of $seed_{benchmark}$ equal to 411193435 on scale factor 40 dataset. Thanks to compression in the DSM storage (see Section 2.4) the total amount of data accessed in the benchmark equals 10 GB. The intra-query parallelization was enabled with the maximum number of threads that can be spawned set to 8. In experiments with Cooperative Scans we used both CScan operators and the Cooperative Merge Join, so that the clustering of tables could be exploited.

We ran the TPC-H throughput benchmark on a server equipped with two quad-core Intel(R) Xeon(R) X5560 CPU and 48 GB of RAM memory. It is enough memory to be

able to cache the whole TPC-H scale factor 40 dataset in the buffer pool. Also, the remaining memory suffices for the query memory used for data structures needed for processing of queries such as hash tables. The I/O subsystem is comprised of 16 SSD drives connected by means of a RAID-O array. The setup is capable of providing up to 2 GB/s of bandwidth.

7.2.2. Performance evaluation

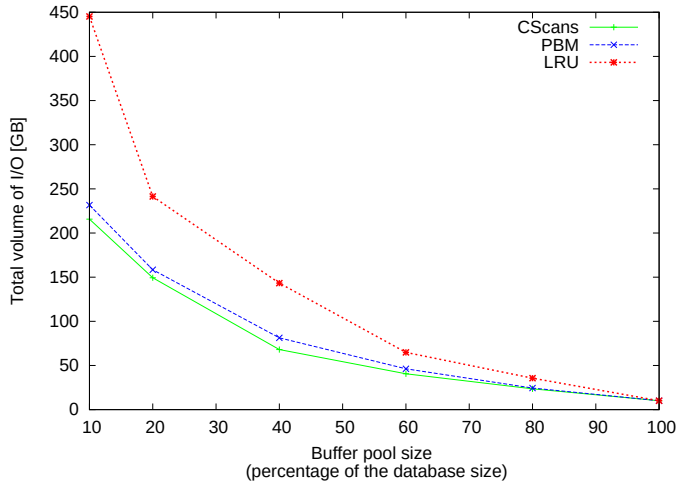
In Figure 7.3 we can see three system-wide result metrics of the TPC-H throughput benchmark, that was run with varying size of the buffer pool ranging from 10% to 100%. We provide the total volume of performed I/O, as well as the average stream time and the TPC-H QphH@40GB metric. The Query-per-Hour Performance Metric (QphH@Size) is calculated as the number of queries that can be processed by the system in an hour, multiplied by the scale factor parameter [Tra11].

Figure 7.3 (a) shows the volume of I/O performed. We can observe that lowering the size of available memory in the buffer pool results in increase of performed I/O for all evaluated solutions. The buffer using LRU policy performs most I/O in all cases, the difference with two other policies is significant for buffer pools of 40% and smaller. On contrary to what we observed in Chapter 6, the advantage of Cooperative Scans over the Predictive Buffer Manager is low. Interestingly, the difference in the volume of I/O performed does not directly translate into lower performance as perceived by the user. Both the average stream time (Figure 7.3 (b)) and the QphH@40GB metric (Figure 7.3 (c)) remain roughly constant for buffer pools capacities from 40% to 100% of the total data volume. Despite 15 times higher amount of I/O operations for the LRU policy and 10 times increase for Cooperative Scans and the PBM, the performance remains virtually unchanged. It is caused by the fact that the system is mostly CPU-bound in those experiments. A very fast I/O subsystem is capable of feeding the query execution layer at pace enough to use its maximal processing speed. This situation changes for smaller buffer pools. In the case of buffer pool capacity of 10% and 20% the need for I/O increases even further and the performance starts to drop. It appears that at this point the vastly increased number of I/O operations causes the query execution layer to wait for data making the system I/O bound. In this case we observe the advantage of Cooperative Scans and the PBM in terms of the average stream time and the QphH@40GB metric. It confirms that it is worth to optimize the I/O performance despite a very fast I/O subsystem. However, the difference in performance is significantly lower that in experiments in Chapter 6.

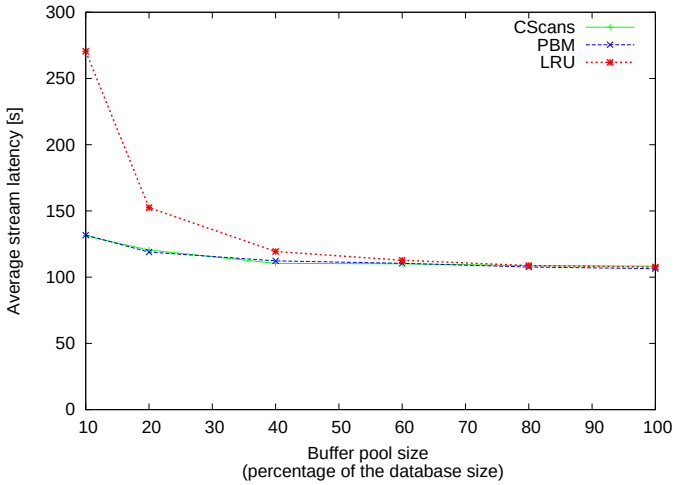
7.2.3. Sharing potential in TPC-H experiments

Figure 7.3 (d) depicts sharing potential analysis analogous to the one found in Figure 6.1. Please refer to Section 6.2 for a description of how the sharing potential plot is generated. The data presented in the plot was gathered in an experiment which used Cooperative Scans.

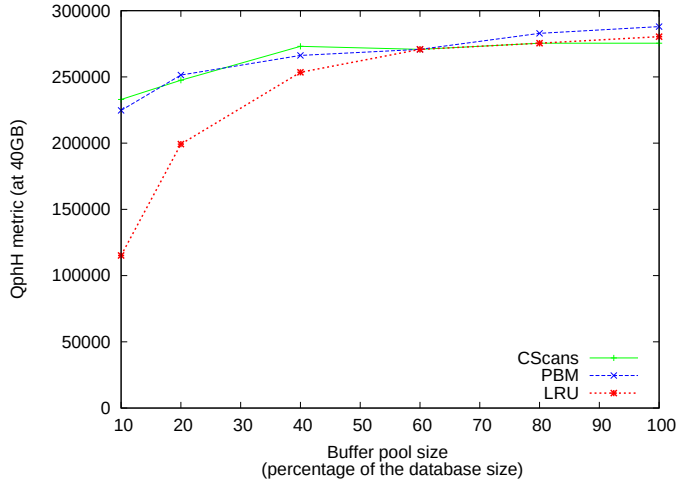
We observe that Figure 7.3 (d) is dominated by green and yellow areas that correspond to the data requested by only one Scan or requested by exactly 2 Scans respectively. Thus, despite the fact that there are 10 streams running in parallel, issued queries access mostly non-overlapping sets of pages. It shows that sharing opportunities in the TPC-H benchmark are low. First, the schema consists of 8 tables instead of 1 or 2 as in the case of synthetic benchmarks presented in Chapter 6. Moreover, as discussed in Section 2.4, queries can share data only if they scan overlapping sets of columns. The sharing opportunities would be potentially much higher in a row-oriented database. Nevertheless, we stress that the benefits of a column-oriented DBMS over row-oriented systems in analytical workloads [AMH08] suggest



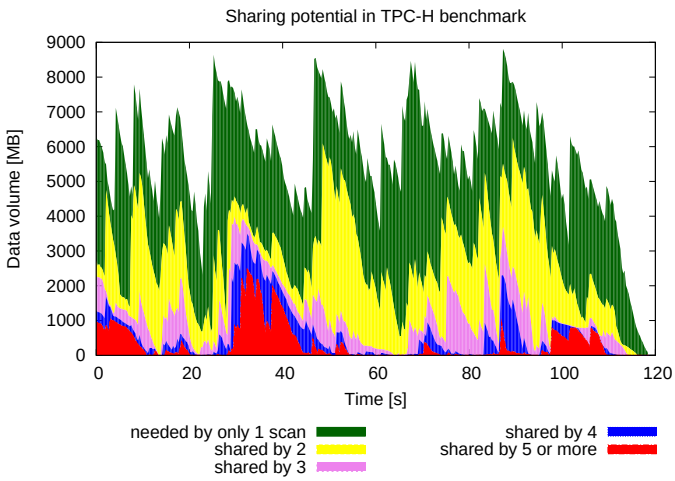
(a) Total volume of I/O



(b) Average stream time



(c) QphH



(d) Sharing potential

Figure 7.3: Performance achieved by different buffer management solution in the TPC-H throughput benchmark

that it is advantageous to use a column-store regardless of lower performance advantage of Cooperative Scans in DSM.

The direct consequence of low sharing potential in the TPC-H throughput is lower performance advantage of Cooperative Scans. As it turned out, the results show the advantage of Cooperative Scans over the LRU policy, but the Predictive Buffer Manager performs equally well. It suggests that the LRU buffer manager misses sharing opportunities that are exploited by both the Cooperative Scans and the PBM. However, there is not enough sharing potential for the Cooperative Scans to prove its advantage over the PBM. As a consequence, it suffices to implement a much simpler solution like the Predictive Buffer Manager in the VectorWise DBMS to improve its performance in the TPC-H benchmark in scenarios with the dataset much larger than the memory reserved for the buffer pool.

7.3. Summary

In this chapter we evaluated the performance of all buffer management solutions discussed in this thesis with the TPC-H benchmark. Together with experiments described in Chapter 6 it constitutes the complete evaluation of solutions proposed in this thesis. We conclude the results in the next chapter.

Chapter 8

Conclusions and future work

8.1. Conclusions

In this master thesis we presented the implementation of Cooperative Scans in VectorWise. We also extended the framework to support order-aware operators. In particular, a novel algorithm Cooperative Merge Join was implemented. The presented implementation addressed all challenges related to introduction of Cooperative Scans in VectorWise that were introduced in Chapter 2 and listed in Section 2.12. Moreover, we proposed and implemented the Predictive Buffer Manager (PBM). It is an alternative solution to Cooperative Scans, that provides some of its benefits and is significantly less complicated.

We also measured the performance of the presented solution using a set of custom synthetic benchmarks and the TPC-H throughput benchmark. Both Cooperative Scans and the Predictive Buffer Manager proved to provide a significant improvement over the standard buffer manager in VectorWise.

The relative performance gain depends strongly on the level of buffer reuse. The possible improvement of buffer reuse is determined by the sharing potential i.e. the volume of data that queries access in common.

Benchmark results show that the potential of Cooperative Scans can be exploited in workloads with high sharing potential. Only then, the Predictive Buffer Manager is outperformed by Cooperative Scans. As a result, from the product development perspective the Predictive Buffer Manager might be preferred.

8.2. Future work

8.2.1. Multiple replicas

VectorWise as a column-oriented DBMS does not provide a fast way to perform small look up queries. To retrieve a single tuple, or a small number of tuples several disk pages have to accessed. To avoid that, a copy of certain tables can be stored in a format allowing fast look ups. Two main possibilities are traditional row storage and hybrid PAX storage [ADHS01]. The idea of combining both column storage and row storage has been researched in [RDS03]. Also, the same table can be stored in column storage in multiple copies with different ordering to be exploited by the query execution layer [SAB+05]. Systems using such hybrid solutions can be equipped with Cooperative Scans as well. The crucial problem to solve is the decision which copies of a certain table should be handled by Cooperative Scans. Depending on the workload, several solutions may be applied. Firstly, Cooperative Scans could be used on copies

that store multiple columns in single disk page, and consequently exhibit increased sharing opportunities. Alternatively, a more sophisticated solution is using Cooperative Scans on all copies. In this case a more dynamic scheme can be created, where the same query would access data from different copies of the same table. For example, a query accessing a small subset of columns, thus using DSM storage by default could also benefit from already-loaded data belonging to a copy stored in PAX format.

8.2.2. Cooperative XchangeScan

Section 4.8 discussed issues related to Cooperative Scans and intra-query parallelism. To solve both the problem of data skew caused by static ranges and different processing speed of threads, we propose a new scheme called Cooperative Xchange Scan. In this approach all CScan operators that belong to the same parallel query plan would be registered in the Active Buffer Manager as a single group scanning specified range. The ABM can exploit the information about CScans belonging to the same query plan. First of all, the ABM could provide better scheduling of loading data. In particular, the ABM would be capable of detecting how many chunks are available for the whole parallelized subtree. Only then chunks can be distributed accordingly minimizing data skew. In simplest case, the distribution can be performed on chunk-at-time level. However, for shorter queries another approach may be needed. In another solution, the input is dynamically split and distributed on demand, which introduces additional overhead, but tackles the problem of load imbalances.

8.2.3. Query scheduling in the Predictive Buffer Manager

As discussed in Section 6.1 the Predictive Buffer Manager in VectorWise suffers from the same problem with query scheduling as the LRU-based buffer manager. This problem has been solved in Cooperative Scans by using the `CScanRelevance()` function (see Figure 3.3). Similar ideas can be incorporated into the PBM. One possible solution would be to follow the approach used in the Active Buffer Manager. The process of issuing requests for I/O operations can be moved from individual queries to the buffer manager, since it is aware of all data that is needed. Then, the PBM could prioritize shorter queries and schedule I/O operations for them earlier. Another, simpler idea is to introduce a system-wide limit for asynchronous I/O requests that can be issued in the system. In this solution, short or fast queries would be allowed to issue more requests than the long and slow queries. By doing so, the data for prioritized queries would be loaded earlier. As a result, short queries may find the data already loaded at the time they issue a blocking request for locking pages that are about to be consumed, whereas long queries may still wait for the loading to be finished.

8.2.4. Throttling of queries in the Predictive Buffer Manager

IBM research proposes another solution for query scheduling based on throttling [LBM+07, LBMW07] of queries that is aimed at increasing buffer locality. This idea can be combined with the Predictive Buffer Manager. The benchmark results presented in Chapter 6 suggest that the performance of the PBM drops when the systems is loaded with many queries scanning at different positions of the table. In such a scenario the PBM is not able to exploit sharing opportunities due to lack of a mechanism „attaching" scans to make them share loaded data. This problem is solved in [LBM+07, LBMW07] by throttling some queries, so that groups of queries scanning at close positions are formed. It may slow down the throttled queries, but at the same time improve the overall system throughput by increasing locality of page requests. The algorithm that forms groups of queries can be extended to

take information used by the PBM into account. Suppose the PBM keeps track of the next consumption time of pages that were lastly evicted from the buffer pool. Let us denote it by *next_consumption_evict*. A page whose next consumption time is higher or close to *next_consumption_evict* is likely to be evicted without being reused. In the PBM every page that has just been consumed is assigned a new next consumption time. By comparing it to *next_consumption_evict* it is possible to detect whether the consumed page is likely to be evicted or reused again. A scan can be throttled when doing so would lower the next consumption time of pages it recently consumed to a value below *next_consumption_evict*. This would allow scans working behind this particular scan to catch up and benefit from pages loaded for the throttled scan.

8.2.5. Opportunistic Cooperative Scans

The Cooperative Scans framework assumes that all loading and evicting decisions are strictly managed by the Active Buffer Manager on chunk at a time basis. This assumption makes the implementation more difficult as the ABM needs to manage much information about the state of the system and keep it consistent. A simpler approach that could benefit from the out-of-order delivery would be to dynamically change the area of table that is processed by a certain query depending on the state of the buffer pool. The Scan operator could constantly monitor which parts of the scanned table contain most cached pages. When a certain region of a table, that is cached to a large extend, is detected the Scan could dynamically change the range to scan that region and possibly increase the number of times those pages are reused before being evicted. In that way scans may automatically „attach" to itself and cooperate.

Acknowledgements

I wish to thank my supervisors Marcin Żukowski and Peter Boncz, who encouraged and challenged me throughout work on this thesis. Thanks to them I was able to deepen my knowledge, explore and utilize my skills and discover databases as a very interesting research area.

Especially, it was Marcin Żukowski, who made this project possible by offering my an internship at VectorWise.

I would like to acknowledge my second reader Frank Seinstra for contributing to scientific aspects of this thesis.

At VectorWise I had an opportunity to work in a friendly atmosphere. I would like to thank Giel, Gosia, Hui, Irish, Michał, Sandor and Willem for answering all my questions and giving me helpful advice. They are a team it is a pleasure to work with.

I thank Kamil, who worked as an intern at VectorWise last year and was always willing to share his knowledge and experience with me.

Special thanks to my fellow co-interns Ala and Julek. We had a great time together at the university, at VectorWise and outside work.

I owe my deepest gratitude to Beata, who convinced me to come to Amsterdam and participate in the Short Track Master's Programme at the VU University. She also gave me tremendous support throughout the whole year. Despite the distance we always had great contact that allowed us both to develop and discover each other.

Lastly, I am grateful to my parents, my sister and the whole family for motivation and encouragement in my education, as well as support in any respect.

Bibliography

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24:1–10, May 1995.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5:78–101, June 1966.

[BZN05] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[CB74] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, SIGFIDET '74, pages 249–264, New York, NY, USA, 1974. ACM.

[CD85] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th international conference on Very Large Data Bases - Volume 11*, VLDB '85, pages 127–141. VLDB Endowment, 1985.

[CK85a] A. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, Austin, TX, USA, 1985.

[CK85b] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, SIGMOD '85, pages 268–279, New York, NY, USA, 1985. ACM.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[Coo01] C. Cook. *Database Architecture: The Storage Engine*, July 2001. `http://msdn.microsoft.com/library`.

[CPV09] George Candea, Neoklis Polyzotis, and Radek Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *Proc. VLDB Endow.*, 2:277–288, August 2009.

[CR93] Chung-Min Chen and Nick Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 342–353, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[Fer94] Phillip M. Fernandez. Red brick warehouse: a read-mostly rdbms for open smp platforms. *SIGMOD Rec.*, 23:492–, May 1994.

[FNS91] Christos Faloutsos, Raymond T. Ng, and Timos K. Sellis. Predictive load control for flexible buffer allocation. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB '91, pages 265–274, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[Gra90] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, SIGMOD '90, pages 102–111, New York, NY, USA, 1990. ACM.

[Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[GUM+10] Georgios Giannikis, Philipp Unterbrunner, Jeremy Meyer, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Crescando. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1227–1230, New York, NY, USA, 2010. ACM.

[HSA05] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 383–394, New York, NY, USA, 2005. ACM.

[HZN+10] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter Boncz. Positional update handling in column stores. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 543–554, New York, NY, USA, 2010. ACM.

[LBM+07] Christian A. Lang, Bishwaranjan Bhattacharjee, Timothy Malkemus, Sriram Padmanabhan, and Kwai Wong. Increasing buffer-locality for multiple relational table scans through grouping and throttling. In *ICDE'07*, pages 1136–1145, 2007.

[LBMW07] Christian A. Lang, Bishwaranjan Bhattacharjee, Tim Malkemus, and Kwai Wong. Increasing buffer-locality for multiple index based scans through intelligent placement and index scan speed control. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1298–1309. VLDB Endowment, 2007.

[RDS03] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. *The VLDB Journal*, 12:89–101, August 2003.

[Roy10] Benjamin Van Roy. A short proof of optimality for the min cache replacement algorithm. Technical report, Stanford University, December 2010.

[SAB+05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[TP71] Toby J. Teorey and Tad B. Pinkerton. A comparative analysis of disk scheduling policies. In *Proceedings of the third ACM symposium on Operating systems principles*, SOSP '71, pages 114–, New York, NY, USA, 1971. ACM.

[Tra11] Transaction Processing Performance Council. *TPC Benchmark H, Revision 2.14.2*, June 2011.

[UGA+09] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2:706–717, August 2009.

[Val87] Patrick Valduriez. Join indices. *ACM Trans. Database Syst.*, 12:218–246, June 1987.

[ZBK04] Marcin Zukowski, Peter A. Boncz, and Martin L. Kersten. Cooperative scans. Technical Report INS-E0411, CWI, December 2004.

[ZHNB06] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. ICDE*, Atlanta, GA, USA, 2006.

[ZHNB07] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative scans: dynamic bandwidth sharing in a dbms. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 723–734. VLDB Endowment, 2007.

[Zuk09] Marcin Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. Ph.D. Thesis, Universiteit van Amsterdam, Sep 2009.