Future Fetch

Towards a ticket-based data access for secondary storage in database systems

Demian E. Vöhringer and Klaus Meyer-Wegener

FAU Erlangen-Nuremberg, Chair for Computer Science 6 (Data Management)
{firstname.surname}@fau.de, https://www.cs6.tf.fau.de/

Abstract. When accessing data from a database, a database management system has to accomplish many tasks. Checking the conformity of the query, generating a reasonably good query plan and executing it in a concurrent system are just a few of them. During query execution, there are some points at which the main execution thread must stop and wait for data. This synchronous waiting can have a major impact on the overall query performance. In this paper we introduce the idea of a ticket-based fetch, which supports the database management system by waiting for data asynchronously. This can improve the query-execution performance in database systems.

Keywords: Asynchronism · Data Flow · Data Access · Ticket · Database System · Data Warehouse · Big Data · Concurrent

1 Introduction

Large database systems (DBS) depend heavily on fast access to huge volumes of data, which can easily exceed the capacity of a server's random-access memory (RAM) by orders of magnitude.

These data can be stored in three different storage spaces, the primary, secondary and archive storage space [9]. They all have different properties that allow them to be used for different tasks. Primary storage uses RAM, while solid-state and hard-disk drives are found in secondary storage, which is orders of magnitude slower. If the data a query is to be executed on is stored in the secondary storage space, they must be transferred to the much faster primary storage space to perform operations on them. It is therefore crucial that algorithms of the DBS depend as little as possible on data in slower storage spaces and try to rely only on data that has already been loaded into primary storage. However, if data are not in main memory, they still must be fetched from disk, which can cause long latencies and slows down the execution of queries.

G. Graefe [6] summarized ideas like indices, buffer management and parallelism in DBS, which allow to cope with these latencies. With the help of indices,

Copyright © 2020 by the paper's authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2738/LWDA2020_paper_2.pdf

data can be found much faster on secondary and archival storage, while good buffer management helps keeping the right data in main memory. Parallelism in database systems is divided in three categories, namely interquery parallelism, horizontal interoperator parallelism, and vertical interoperator parallelism. With interquery parallelism, a DBS can execute several queries simultaneously and therefore use waiting times for data loading by switching to another query that already has its data. Horizontal interoperator parallelism performs the same operation on several partitions of the tuple set at the same time and can take advantage of modern multi-core CPUs. Vertical interoperator parallelism executes operators in pipelines, as shown in [10].

These ideas about parallelism focus more or less on data already accessible in primary storage, or on finding data in the storage spaces, ignoring the fact that the transfer of these data to primary storage adds latency to the execution of each single query. The question is whether we can support these parallelism ideas even better by providing an asynchronous data flow inside a DBS.

This can be useful in a query that connects many tuples from different tables to create one large result tuple, as it is done in star-shaped queries (e. g. Listing 1.1). A star-shaped query is similar to a star query, which uses a main table (fact table) that contains information about how to connect the other tables (dimension tables). The only difference is that star queries are bound to data-warehouse systems and use information from the dimension tables mostly for reducing the result-set size[1]. With a limited number of values in the dimension tables, star queries can be optimized by using bitmap indices, while star-shaped queries cannot use this kind of optimization and are thus much harder to optimize.

In this paper we present some first steps of the idea of a ticket-based data loading from secondary storage. We do this by introducing a ticket system for database operators. It desynchronizes data and execution flow of query processing.

We begin our journey with a simple example. We describe the problem (Section 2), find a first solution (Section 2.1), explain that solution in depth (Section 2.2), and try to expand that solution, so it can be executed in a more general workflow of query execution (Section 2.3). After that we discuss related work (Section 3), draw a conclusion, and present future work (Section 4).

2 The Idea

To work with the data, a classical DBS must transfer it from secondary or archival storage to primary storage. This process is shown in Figure 1a for a star-shaped query. Because the data-tranfer rate of secondary storage is limited, access to the data takes some time, so any query execution in need of these data must interrupt its execution flow and wait some time for the data to arrive.

This pause can become large depending on the exact location on disk and the access path for it, and the query execution must wait, even if the data is

[1] `http://www.orafaq.com/tuningguide/star%20query.html`

not needed immediately. To improve the search, indices have been introduced, which can help to find the required data faster on secondary storage. Once the data is collected, query execution proceeds to the next step, which may require some more data. These steps must be repeated for each tuple-fetch operation, until the result set is complete and can be returned to the caller. So waiting for data can take a large portion of the execution time of a single query.

To reduce this data latency, we present an initial idea of ticket-based data access, describe it in depth, and expand it towards a more general query execution.

2.1 A first step

To make the idea understandable, we simplify the problem a bit. We ignore some ideas on parallelism presented in [6] and only focus on one query that works on one large set of tuples. This way we still have vertical interoperator parallelism, but neither interquery parallelism nor horizontal interoperator parallelism. Additionally we assume that secondary indices (an index not containing data, but only information where to find the data in the storage spaces) can be kept in primary storage for fast access.

With our idea we want to intervene exactly where the query execution must wait for data and try to improve the execution time with asynchronous data access. As shown in Figure 1c, our first step towards asynchronous data access focuses on an index scan (a scan operator using an index) combined with a special *Data Access Manager*, our abstraction for access to secondary storage and moving data to primary storage.

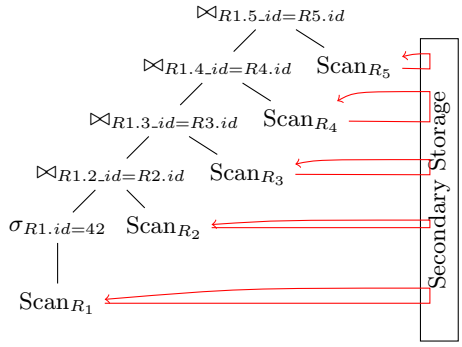
Traditionally, an index scan performs three steps in accessing the data from secondary storage. First it uses a secondary index and checks whether and where the corresponding tuple can be found. Because of our assumption, this is done in main memory and thus very quickly. So the location of the tuple on secondary storage is known. In the second step, this tuple is loaded to primary storage, where the third step continues and returns the tuple to the calling operator. In our example, this is a join operator.

If we now rework this index-scan operator to use the Data Access Manager, we obtain two operators. We get a "check for existence, get the storage-space address and start loading" ($\exists$) and a "wait for loading and combine the loaded tuple with the existing tuple" (∞) operator, as seen in Figure 1b. We describe these operators and the Data Access Manager in detail in Section 2.2. You can easily see that the second step has been delegated to the Data Access Manager.

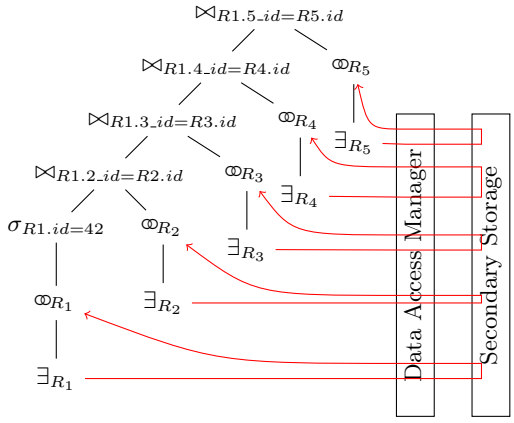
Now that we have refactored our index scan into two operators, we move them in our query-execution tree to optimize it, as shown in Figure 1c. The $\exists$ operator remains in the same place where the scan was, while the ∞ operator is moved to where the data is actually needed. In our example, this is the end of query execution just before returning the result tuple, where we insert the "wait and combine all data" operator. Because the data from R_1 are needed right now, we can use the old scan operator for simplicity.

```
SELECT *
FROM R1, R2, R3, R4, R5
WHERE R1.id = 42
AND R1.2_id = R2.id
AND R1.3_id = R3.id
AND R1.4_id = R4.id
AND R1.5_id = R5.id;
```

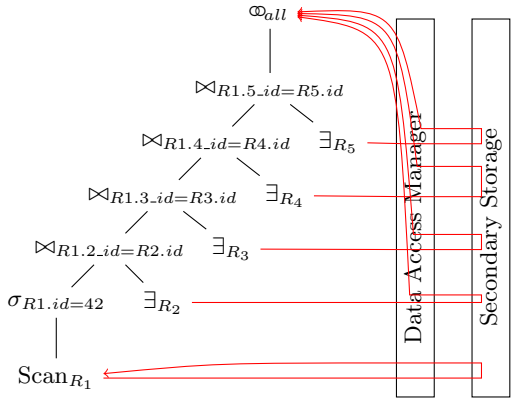
Listing 1.1: Exemplary SQL for a star-shaped query



(a) Traditional access approach



(b) Traditional access approach with our newly designed operators and the Data Access Manager



(c) The complete ticket-based system approach

Fig. 1: Execution trees of the star-shaped query presented in Listing 1.1. Access to data from secondary storage is presented in red.

We can now easily see that the main execution path just informs the Data Access Manager which tuples need to be moved from secondary storage to primary storage for easy access. This has several advantages, as described in the details (Section 2.2).

We expect that this change in strategy can already improve the execution time of quite small star-shaped queries in huge DBS, and we hope to show in the future that this also holds for other query types.

2.2 The operators in detail

Here we explain in more detail our thoughts on the two operators and the Data Access Manager mentioned above.

The $\exists$ operator is designed to be called with a table and an expression (e. g. simple search argument). It queries a secondary index structure for the position of the tuple in secondary storage space. With this address and a certain priority, it calls the Data Access Manager and receives a *ticket* (e. g. a pointer or "future object"[2]) with information later allowing access to the fetched tuple. This ticket is then forwarded to the next operator. Now the data flow is separated from the execution flow and the current tuple is much smaller, since data not currently needed are not yet part of the tuple. This may have the additional advantage of being able to store the tuple closer to the CPU.

The ticket is now part of the tuple and is resolved by the ∞ operator when the data are needed. The ∞ operator looks at the ticket and determines whether the data is already stored in primary storage. If this is not the case, it recalls the Data Access Manager with the ticket, requests a higher priority and waits for the data to be delivered. This results in a delay that should not be greater than the normal access delay in a synchronous call. Hopefully the data can already be found and can be accessed directly without any delay. Now the data can be combined with the tuple and can be presented to the next operator. If the ∞ operator needs to get the data from multiple tickets at the same time, it can prioritize the already loaded data and can give the Data Access Manager more time for completing the other tickets.

The Data Access Manager has a list of all tuples that must be moved to primary storage and can therefore improve throughput by taking advantage of the secondary storage device's properties, reorganizing the tuples' access order accordingly[3]. If a tuple is needed right now, the access order can be rescheduled to make that data accessible sooner.

2.3 The next steps

The first step exploits some simplifications, like reducing the parallelism and keeping the secondary indices in primary storage. We see no problem in bringing

[2] `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html`

[3] `http://www.cs.iit.edu/~cs561/cs450/disksched/disksched.html`

the two other parallelism ideas back. The secondary indices in primary storage gives us the advantage of knowing how many tuples to expect without needing to access the secondary storage. Additionally we have dropped the idea of batch execution, where at the same time multiple tuples are given to an operator as input.

We now like to expand the idea to overcome these simplifications.

For querying a secondary index that is not in primary storage, we move the querying of the secondary index into the Data Access Manager. The $\exists$ operator then does not give an address to the Data Access Manager, but just the table and the expression. By limiting the information, we force the Data Access Manager to do a normal index scan, where the index might not be stored in primary storage. This way we cannot predict how many tuples are going to be returned. Based on our ticket system, the Data Access Manager now creates a data structure that stores from zero to multiple tuples in it. The $\bowtie$ operator reacts to this by either dropping the given tuple (it has no join partner), or by iterating through the data structure and creating a combination with each of the retrieved tuples, resulting in multiple tuples for the next operator. Depending on the query and the expected drop rate, the query optimization may place the $\bowtie$ operator closer to or farther away from the $\exists$ operator to balance between reduced unnecessary workload and increased parallel execution.

With this change, we can use arbitrary scan operators with the Data Access Manager, and may also be able to include basic operations such as a projection or selection to increase the amount of parallel execution and to reduce the amount of space on primary storage needed to store the tuples.

To handle batch execution, we expect the $\exists$ operator to open a ticket for each of the tuples in the batch. No further adjustments are needed, because the remainder of the execution can be kept as if there was only one tuple, only that we now have to do this for each tuple of the batch. The $\bowtie$ operator has another small advantage in being able to check tickets from multiple tuples at once, rather than waiting for the data of a particular tuple to be accessible right now.

With these steps we expect to be able to handle arbitrary queries in current DBS.

3 State of the Art

We found the idea of increasing the speed of slower devices already in the work of Sarawagi [12], where the optimization of access to tertiary memory (archival storage) is discussed. Here the author suggest using a scheduler to first bring all needed data in the secondary storage space and then start executing the query on it. Even if you do not consider that we are talking about secondary to primary storage data transfer and not archival to secondary, we still differ in the idea, because we try to make data accessible while running the query and not to load the data into a faster storage space before executing the query, which may not need all the data.

Codd described a restriction in [4] that was later used in the concept of semi-joins [2]. This operator exploits the idea of reducing a relation to the tuples needed before joining it with another relation. With this reduction all unneeded tuples are dropped early and the join can be performed much faster. In particular if working over a network, the data to be transferred can be reduced substantially. The idea seems similar to ours, but semi-joins are not using the potential of asynchronous data flow, and the relations are reduced in the execution of a join and not before.

For asynchronous data access there are some implementation concepts like asynchronous I/O[4] or future objects[5]. These are some basic techniques for adding asynchronism to software systems and we plan to use them for implementing the Data Access Manager.

The idea of splitting an operator into smaller bits and looking at these bits was recently proposed by Dittrich and Nix [5]. This paper focuses on the idea of creating physical operators that fit the data better, while we focus on establishing a new asynchronous data flow that ignores the borders of scan operators.

Gurumurthy et al. [7] gave an overview of constructing operators from smaller parts and adapting more easily to new specialized hardware like GPUs [8] and FPGAs [1]. We share the idea of having smaller parts in operators, but focus on data access and not on adaptability to new hardware.

There have been some ideas for operators that can take advantage of better disk-scheduling algorithms, e. g. [3]. Here the authors present a physical scan operator that can adapt to errors in cardinality estimation by prefetching more blocks with tuples to tune between an index and a table scan. We, in contrast, are not defining a new physical operator to replace a logical operator, but change the size and borders of existing scan operators to better optimize the data flow in the query execution.

We found the idea of optimizing star queries inside the Oracle Documentation [11], but this is very specific for star queries in data warehouses. The data from the dimension tables is only used for comparison with constraints and is dropped afterwards. This way they use bitmap indices for the dimension tables to say whether a tuple fits the constraint, which can increase the speed of star queries in data warehouses. We just use star-shaped queries as an easy example and must return the data from the "dimension" tables.

In HyPer [10] a data-aware computing method was introduced by switching the execution direction inside of a pipe from pulling to pushing. By doing this, the data can be kept in CPU registers and need not be stored and loaded from slower storage areas. We do not focus on keeping the data close to the CPU, but instead try to minimize the waiting time when loading the data into a faster storage space. We think it might be possible to combine these two ideas.

[4] `https://man7.org/linux/man-pages/man7/aio.7.html`
[5] `https://en.cppreference.com/w/cpp/thread/future`

4 Conclusion and Future Work

In this paper we present our idea of desynchronizing the data and execution flow in query processing. We do this by introducing a Data Access Manager and by splitting scan operators into new operators focused on data access. This way we are able to place them independently in the query-operator tree. We expect to reduce the latency for data access in a DBS and the overall query-execution time.

We are fully aware that this idea increases the search space of query optimization even more, but we rather see this as a challenge for research than a problem.

As future work we plan to implement and test our idea. Because we expect the rework of the software to be quite large, we have decided to use a lightweight, relatively simple, and open-source database management system, namely Apache Derby[6].

We plan to do this by dividing the task into subtasks, which are: the implementation and evaluation of the Data Access Manager with the ticket system, the splitting of the scan operator, and an additional optimizer step for the physical execution plan that replaces the standard operators with our operators and moves them to fit our idea. Afterwards, we would like to extend the implementation with ideas from our next steps (Section 2.3). For evaluation we plan to run some queries from the TCP-H benchmark on datasets magnitudes larger than the RAM of the specific machine. Furthermore, we think about adding the idea to distributed or federated DBS, to reduce the data-access latency of an access to a remote system.

References

1. Becher, A., B.G., L., Broneske, D., Drewes, T., Gurumurthy, B., Meyer-Wegener, K., Pionteck, T., Saake, G., Teich, J., Wildermann, S.: Integration of fpgas in database management systems: Challenges and opportunities. Datenbank-Spektrum **18**(3), 145–156 (Nov 2018). https://doi.org/10.1007/s13222-018-0294-9
2. Bernstein, P.A., Chiu, D.M.W.: Using semi-joins to solve relational queries. J. ACM **28**(1), 25–40 (Jan 1981). https://doi.org/10.1145/322234.322238
3. Borovica-Gajic, R., Idreos, S., Ailamaki, A., Zukowski, M., Fraser, C.: Smooth scan: robust access path selection without cardinality estimation. The VLDB Journal **27**(4), 521–545 (2018). https://doi.org/10.1007/s00778-018-0507-8
4. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (Jun 1970). https://doi.org/10.1145/362384.362685
5. Dittrich, J., Nix, J.: The case for deep query optimisation. In: CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings. www.cidrdb.org (2020), `http://cidrdb.org/cidr2020/papers/p3-dittrich-cidr20.pdf`
6. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**(2), 73–169 (Jun 1993). https://doi.org/10.1145/152610.152611

[6] https://db.apache.org/derby/

7. Gurumurthy, B., Broneske, D., Drewes, T., Pionteck, T., Saake, G.: Cooking dbms operations using granular primitives. Datenbank-Spektrum **18**(3), 183–193 (Nov 2018). https://doi.org/10.1007/s13222-018-0295-8
8. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query coprocessing on graphics processors. ACM Trans. Database Syst. **34**(4) (Dec 2009). https://doi.org/10.1145/1620585.1620588
9. Kemper, A., Eickler, A.: Speichermedien. In: Datenbanksysteme, pp. 211–212. Oldenbourg Wissenschaftsverlag GmbH (2013), 9th edition
10. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. **4**(9), 539–550 (Jun 2011). https://doi.org/10.14778/2002938.2002940
11. Rich, B.: Oracle database reference, 12c release 1 (12.1) e17615-20 (2017), `https://docs.oracle.com/database/121/DWHSG/schemas.htm#DWHSG9069`
12. Sarawagi, S.: Database systems for efficient access to tertiary memory. In: Proceedings of IEEE 14th Symposium on Mass Storage Systems. pp. 120–126 (1995). https://doi.org/10.1109/MASS.1995.528222