Engineering High-Performance Database Engines

Thomas Neumann
Technische Universität München
Department of Informatics
Munich, Germany
neumann@in.tum.de

ABSTRACT

Developing a database engine is both challenging and rewarding. Database engines are very complex software artifacts that have to scale to large data sizes and large hardware configurations, and developing such systems usually means choosing between different trade-offs at various points of development.

This papers gives a survey over two different database engines, the disk-based SPARQL-processing engine RDF-3X, and the relational main-memory engine HyPer. It discusses the design choices that were made during development, and highlights optimization techniques that are important for both systems.

1. INTRODUCTION

Building a database engine is a very complex task. Besides the pure code size, many decisions must be made that affect trade-offs within the system. As an illustrative example consider the choice between disk-orientation versus main-memory-orientation. If we assume that data is mainly on disk, sequential data access is extremely important, and aggressive compression is a good idea in order to save I/O bandwidth. If, however, data is largely in main-memory, we can afford much more random access, and compression becomes a burden rather than a virtue due to CPU costs. Similar, although usually less severe, choices are made constantly during system development. On the other hand some choices like increasing locality or using a query optimizer are virtually always a good idea. It is therefore instructive to look at existing systems to see what they did right and what could be improved.

In this paper we give a survey of two different systems. One is the disk-oriented system RDF-3X [37] for managing graph-structured RDF data, the second is the main-memory relational DBMS HyPer [16]. Both were developed to a large extent by the author of this paper, but they have radically different architectures. We show why that is the case, and how the different design goals affected the systems. Furthermore, we show which techniques, in particular in the context of query optimization, were essential for both systems and therefore should be applicable to other systems, too.

The rest of the paper is structured as follows: We first describe the disk-based system RDF-3X in Section 2. In Section 3 we then contrast that to the architecture of the relational main-memory DBMS HyPer. Query optimization techniques are discussed in Section 4. Finally, an overview of other systems is given in Section 5.

2. RDF-3X

The RDF (Resource Description Framework) data model has been designed as a flexible representation of schema-relaxable or even schema-free information for the Semantic Web. Nowadays it is used, beyond the original scope of the Semantic Web, for many kinds of graph-structured data. In RDF, all data items are represented in the form of $(subject, predicate, object)$ triples. For example, information about the song “Changing of the Guards” performed by Patti Smith could include the following triples:

$(id1,\ hasType,\ "song")$,
$(id1,\ hasTitle,\ "Changing\ of\ the\ Guards")$,
$(id1,\ performedBy,\ id2)$,
$(id2,\ hasName,\ "Patti\ Smith")$,
$(id2,\ bornOn,\ "December\ 30,\ 1946")$,
$(id2,\ bornIn,\ id3)$,
$(id3,\ hasName,\ "Chicago")$,
$(id3,\ locatedIn,\ id4)$,
$(id4,\ hasName,\ "Illinois")$,
$(id1,\ composedBy,\ id5)$,
$(id5,\ hasName,\ "Bob\ Dylan")$,
and so on.

Conceptually an RDF database is a huge set of such triples, where the triples implicitly form a graph (every triple is an edge in the graph). This graph can be queried using the SPARQL query language. For instance, we can retrieve all performers of songs composed by Bob Dylan by the following SPARQL query:

```
Select ?n Where {
  ?p <hasName> ?n.  ?s <performedBy> ?p.
  ?s <composedBy> ?c.  ?c <hasName> "Bob Dylan"
}
```

For a database system the largely schema-free nature of RDF data is very challenging. Data is stored in one (or

```
 select ?u where {
?u <crime> [].  ?u <likes> "A.C.Doyle";
?u <friend> ?f.
?f <romance> [].  ?f <likes> "J.Austen" .
}
```

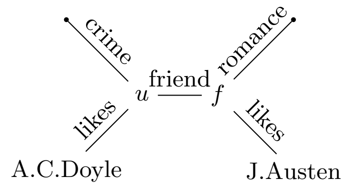

Figure 1: A SPARQL query with a corresponding graph representation.

more) huge graph(s), the edge structure can vary wildly, and both data distributions and queries are hard to predict. Note that while SPARQL queries *usually* form nice subgraphs that are to be matched in the data set (see Figure 1 for a visual example), the user is free to formulate very unusual query shapes, including, for example joins over unbounded predicates. This means that all techniques that rely upon known predicate values are not well suited to handle arbitrary SPARQL queries.

The RDF-3X engine [37, 39] was designed based upon three assumptions:

1. It is impossible to predict the shape of the data graph or the shape of the queries. Therefore, data must be stored and indexed such that any possible kind of SPARQL query can be answered efficiently.

2. The data graph is large, typically larger than main memory. Therefore, query processing will often be I/O bound, and the system must maximize sequential access and minimize I/O bandwidth.

3. Even though the graph is large and queries can touch arbitrary parts of the graph, the result that will be presented to the user is usually reasonably small. Therefore, it pays off to aggressively prune and to delay reconstruction to the end.

Not all of these assumptions are necessarily always true, in particular main memory sizes have increased significantly since the start of RDF-3X development. However, they are plausible, and adhering to them has led to a very scalable and robust system.

Internally, RDF-3X is organized into several layers [37, 39]. On the storage layer, all RDF triples are first passed through a dictionary compressor, which replaces all strings and literals in the triples with dense integer numbers. This greatly simplifies further processing, and also reduces the data size significantly, as the degree of redundancy in the data is often high. These integer triples are then stored in clustered B^+-trees. Clustered B^+-trees are very disk-friendly, both for point accesses and for range queries; and in the context of SPARQL, we can translate every possible query pattern into a single range that immediately returns all matching tuples: The triple pattern `?u <likes> "A.C.Doyle"` can be interpreted as $p = \text{likes} \wedge o = \text{A.C.Doyle}$, or, after dictionary compression, as something like $p = 1 \wedge$

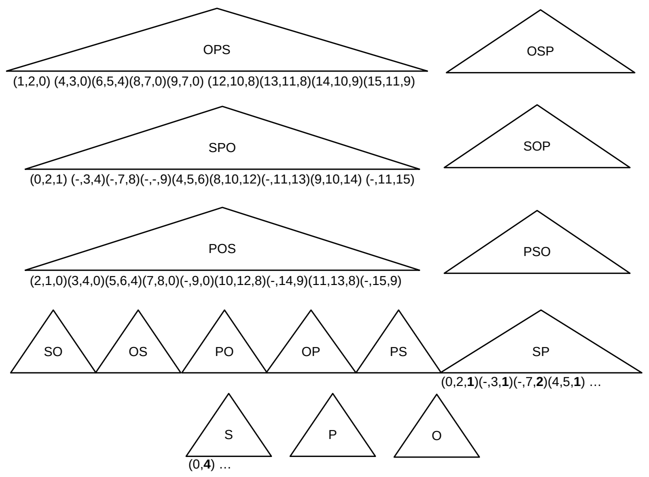

Figure 2: Indexing Triples in RDF-3X

$o = 1234$. Therefore, if we have an index on (P,O), we can answer that pattern immediately with one range scan. Because we make no assumption about the shape of the query, any combination of attributes is useful, and RDF-3X therefore indexes all combination of S, P, and O. The size overhead of this redundant indexing is mitigated by using compressed B-tree leaves, the total storage size is usually less than the original data size. This is shown in Figure 2. The "-" marks repeated values that can be totally omitted, they other values are stored in an tight variable-length representation.

The query processing layer exploits the fact that internally all literals are represented as integers. Even though a literal can have a data type like string or date, typing is dynamic and the type information is stored within the dictionary. The query processing operations usually reason only over integer values. This again is a trade-off, of course. Equality comparison is very cheap due to that design, predicates that require type information are more expensive, as they require consulting the dictionary. But overall this representation leads to a very lean and efficient operator model, which the original paper calls *RISC-style* [37]. In particular, joins are cheap and well supported. The system uses merge joins if the tuple order produced by the indexes accessed is favorable, otherwise it uses hash joins.

RDF-3X's runtime infrastructure depends upon a good query optimizer to find a good execution order. This is very challenging in RDF due to the lack of a schema. In particular, cardinality estimation is very hard. We will discuss this again in Section 4. But overall RDF-3X is quite successful there, and usually finds a very good execution plan. As a result, its initial release outperformed other systems significantly [37, 39], often by a factor of 10 or more.

For very large graphs with billions of triples efficient data access is even more important. A query might be selective in itself, but an individual triple pattern can access significant parts of the database, which then causes prohibitive costs. For RDF-3X, we therefore developed techniques for *sideways information passing* [38] that allows different parts of the query to actively prune observed data against each other, and to restrict the index accesses using this information. The simplest example is a merge join, where nonjoining parts of the data can be skipped in a zig-zag fashion depending on the other side of the join. Similar techniques can be derived for hash joins, and large join groups in general. As a result, RDF-3X often touches only a fraction of the original data, which is very helpful for disk-based systems.

RDF data is usually read-mostly, but of course every RDF store is updated from time to time. For this case, a database engine should offer isolation guarantees, ideally serializability. In RDF-3X transactions are implemented in a multi-step approach [40]: First, changes are kept local and only visible to the writer transaction. At commit, they are merged into differential indexes (relative to the main database) and annotated with version numbers. From time to time the differential indexes are merged back into the main database, updating the B-trees as needed. The versioning greatly simplifies transaction isolation, as each transaction can decide individually if a triple is visible for it or not. As an added benefit, they even allow for time travel queries. The differential indexes are important due to the disk-orientation of RDF-3X: extensive indexing of all attribute combination virtually guarantees that at least some indexes will have poor locality for a given update transaction. By caching them in small intermediate indexes the random updates are aggregated into more sequential merges. A nice property of SPARQL is that it is relatively simple to implement predicate locking, which means that RDF-3X can offer fine-grained locking without excessive locking costs.

In retrospective, the architecture of RDF-3X proved to be very robust and scalable, but was perhaps geared a bit too much towards the on-disk case. It was initiated when large RDF data sets were still much larger than main memory, but that is about to change. And if the data is already in main memory, decompression, which is very useful to save disk I/O, causes a significant CPU overhead. Consequently, some carefully scheduled random accesses would be very useful to improve performance for the in-memory case, but would be prohibitive expensive in the on-disk case. So today some re-balancing of the trade-offs might be a good idea. On the other hand these are largely tuning and implementation issues, they do not require a complete rewrite. And the original premise of assuming nearly nothing about the data and the queries was a very good idea. It has led to a system that is very robust and can handle various kinds of use cases while offering excellent performance.

3. HYPER

Where RDF-3X is disk-oriented, the newer HyPer system is a relational main-memory database system [16]. In the last years main memory sizes have grown to a point where servers with more than a terabyte of RAM are affordable, which means that the transactional data of most companies can fit into main-memory of a single server. Additionally, main memory offers not only faster I/O than disk, but comes with hardware support for interesting features. In particular, main memory databases offer a chance to close a rift that has developed in the engine landscape: Historically, database engines are either good at online transaction processing (OLTP) or online analytical processing (OLAP), but usually not at both. This rift has developed out of technical necessities, not out of user requirements. Users would often like to run complex analytics on the live OLTP system, too, but without hurting mission-critical OLTP performance. This was the original motivation for the development of HyPer: Exploit main-memory technology to develop a unified OLTP and OLAP system that can do both OLTP and OLAP *efficiently* and *simultaneously*.

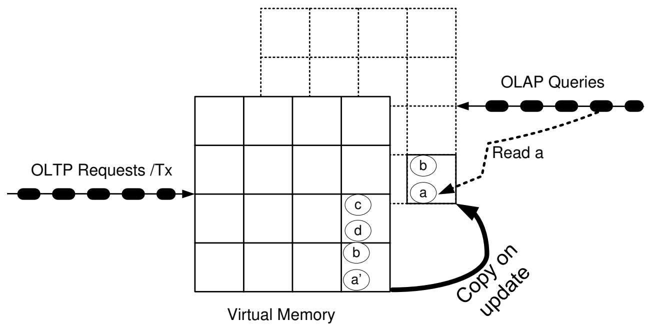

Figure 3: Transaction Isolation using Virtual Memory Snapshots

These observations have led to the development of the HyPer system [16, 17], which was designed based upon these assumptions:

1. Running OLTP and OLAP simultaneously is highly desirable. Therefore, a system has to ensure that they do not hinder each other. For example, a long-running OLAP query must not interfere with concurrent OLTP transactions.

2. Main memory is so large that the working set of transaction processing fits into main memory. Therefore, transactions never block due to external I/O.

3. Another source of blocking behavior could be transaction isolation. Therefore using copy-on-write virtual memory support to provide non-blocking transaction isolation is highly desirable.

4. If transactions never block, any CPU overhead is immediately visible. The system must therefore be heavily tuned to minimize CPU costs, much more than in a disk-based system.

One of the original ideas of HyPer was that if data is in main memory, we can use virtual memory to provide transaction isolation at low costs [16]. This is illustrated in Figure 3. The "main" database, that executes OLTP transaction as quickly as possible, is copied in a virtual manner for incoming OLAP transactions by using the *fork* system call. After the fork, both copies are physically the same, but virtually separate, offering a stable snapshot at low costs. The moment a shared page is modified, the memory management unit traps into the operating system and creates a copy of the page, hiding the change from the other transaction. The nice thing is that we get transaction isolation at a very low cost. The only noticeable cost is the one-time costs of the fork call. Within the transactions, we do not need any kind of latching or other synchronization mechanism, at least as long as we execute the OLTP serially, as proposed by VoltDB [14]. For performance reasons HyPer deviates from serial execution if it can decide that transactions operate on disjoint parts of the data, but it never uses locking or latching. Experiments have shown that classical 2PL is so expensive to implement that serial execution is

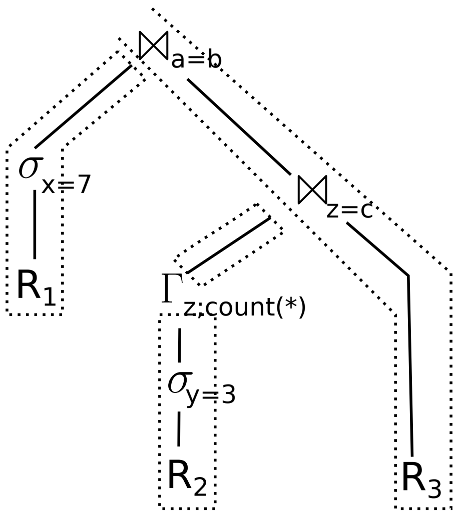

Figure 4: An Execution Plan with visible Pipeline Fragments

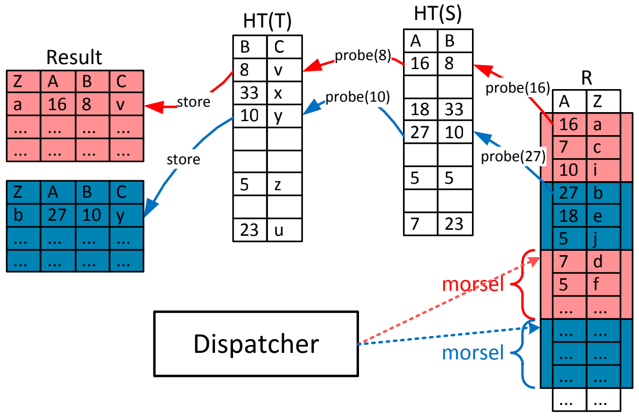

Figure 5: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$

much more attractive in a main-memory OLTP system, as it can use the CPU at full speed and never waits.

On recent processors, namely those that support hardware transactional memory (HTM), it becomes feasible to deviate from serial execution without paying the high costs of latching and locking [22]. There, multiple threads can work optimistically on the same data, only in the unlikely event that they actually collide does the system fall back to a lock to synchronize access. Note, however, that while HTM is enough to provide thread synchronization, it is not enough for transaction isolation. Database transactions are too complex for current HTM systems, thus the database has to assemble a database transaction from multiple hardware transactions [22]. Similar, the database needs a mechanism to bring changes from the virtual snapshots back into the main database. This is not an issue for the more common read-only OLAP queries, but if OLAP transactions decide to modify the data, the database can use a similar mechanism to decide if changes can be safely merged back into the main database [27].

All this assumes that transactions, in particular OLTP transactions, are very fast. HyPer achieves this by using LLVM for just-in-time compilation of SQL queries to machine code [32, 34]. The compilation works in a data-centric manner: Instead of compiling one operator at a time, the compiler generates code for each individual data pipeline of the execution plan. This is illustrated in Figure 4. The pipelines, i.e., the data flow paths from one materialization point to the next, are compiled in a bottom-up manner where every operator pushes tuples towards its consumer. In code this means that most pipeline fragments consist of a few tight loops, which is favorable for modern CPUs and results in excellent performance. This compilation step avoids the high CPU overhead of classical interpreted execution frameworks. For disk-based systems this overhead was largely neglectable, but for in-memory processing any interpretation overhead is very visible [32].

Similar, HyPer uses a specialized data structure for in-memory indexing [21]. Disk-based systems traditionally use B-trees, and for good reasons, as B-trees are a well proven data structure that offers good on-disk performance. Even in memory B-trees are not a bad choice, and perform much better than for example red-black trees, which are traditionally used for in-memory search trees. The reason for that is that red-black trees are much deeper than B-trees, which is very expensive due to caching and mispredictions on modern CPUs. The main disadvantage of B-trees is that they are still comparison based, and comparisons (and the subsequent unpredictable branch) are very expensive nowadays. HyPer therefore use radix-trees to avoid expensive comparisons, and uses an adaptive data layout to avoid the space overhead typically associated with radix trees [21]. This results in a data structure with very good performance, and which improved for example TPC-C transaction rates in experiments by over 70%.

Very good single-threaded performance is good, but not sufficient on modern multi-core systems. Systems with 60 cores and more are becoming common, and a system must strive to utilize all of them. Therefore massively parallel multi-core algorithms that take NUMA into account are being developed, such as our MPSM [1] massive parallel sort merge join, or hash based join algorithms [20, 2]. On the other hand system utilization can be hard to predict, which makes over-committing to a large number of threads dangerous. HyPer therefore always uses a fixed number of threads (one for each core), and multiplexes the workload to the worker threads [20]. In order to increase locality, in particular in NUMA settings, the worker threads first try to process local work in small morsels, and only operate on remote data if there is no local work left. This adaptive work stealing allows for very flexible execution, and in particular allows for adjusting the degree of parallelism dynamically during execution. This task is carried out by the *Dispatcher* module which assigns workers a small piece of work (a *morsel* of about 100,000 tuples) at a time – as exhibited in Figure 5.

Even though main-memory sizes are growing, HyPer has to care about large data volumes, of course. In particular in warehousing scenarios data can go beyond main-memory, and simply storing it as it is in main-memory is not enough. One useful technique here is database compaction [7], where the database separates the hot data, i.e., the data that is accessed (and potentially updated) very frequently, and the cold data that is mainly kept for archival and accessed more rarely. This separation is done by observing access patterns, and again virtual memory can be exploited for that. After separation it pays off to compress cold data aggressively, and it will most likely never change again, which saves space and also makes it reasonable to spool the data to disk. Similar disk-based issues arise if data is imported from external sources in the form of text files. Parsing and importing text files is surprisingly expensive when handling huge volumes of data. HyPer therefore uses a number of techniques like using SSE instructions for parsing and careful parallelization of the import process to allow for "*instant loading*" [29]. Instant here means that the system can import the data as fast as it is delivered from the network or very fast disks (e.g., 10GbE or SSDs). This allows for query processing at wire speed.

Another way to address very large data volumes is scale-out, i.e., distributing the data over multiple nodes. This works particularly well if it is possible to distribute the queries, but to keep the transaction processing largely on one node [28]. Distributing data however creates a whole new class of problems, as network transfer is expensive. This can be mitigated to some extend by recognizing locality that often occurs naturally in the data [43]. For example, if data is stored in time-of-creation order, neighboring tuples will have very similar date and time information, which can be exploited to reduce the volume of network traffic.

Overall HyPer offers excellent performance, not only in OLTP settings like TPC-C or in OLAP settings like TPC-H, but in both simultaneously. That was traditionally not possible within a single database engine. The combined processing functionality has inspired people to consider combined OLTP and OLAP benchmarks, as in the mixed CH benchmark [5]. It combines a transaction mix from TPC-C with a query stream inspired from TPC-H running on the same data, and thus stresses combined processing functionality. This certainly seems to be a promising direction; use cases like business intelligence simply need the functionality to query live OLTP data and databases should offer this functionality.

4. QUERY OPTIMIZATION

Both RDF-3X and HyPer rely upon query optimization techniques for finding good execution plans. Query optimization is absolutly essential for virtually any database system that has to cope with reasonably complex queries. As such, it always pays off to invest time in the optimizer. Often, the impact of the query optimizer is much larger than the impact of the runtime system!

This is illustrated in Figure 6. It shows the execution times for TPC-H query 5 (SF1) in HyPer when using 100 random join orders (with a timeout after 3 seconds) [42]. HyPer compiles the query into machine code, parallelizes its execution, uses semi-join techniques, and uses all kinds of performance tricks to be fast, but it still cannot mitigate the devastating effect of picking a bad join order. The difference between the best plan and the worst plan in that figure is more than a factor of 100, and the plan that the HyPer optimizer picks is in fact even better than the best plan shown there. The runtime system has often no chance to correct the mistakes made by the query compiler. This makes a good query optimizer essential.

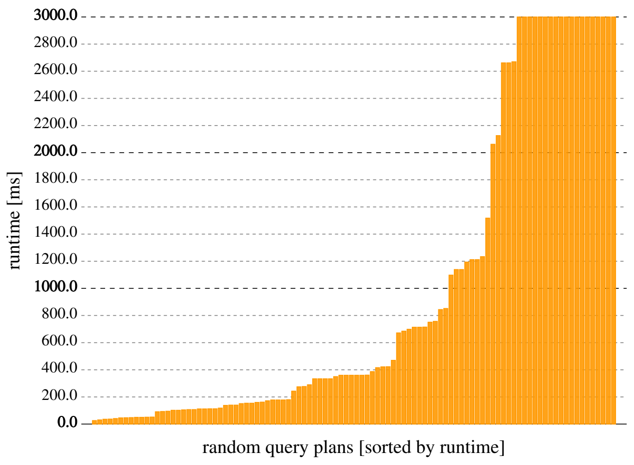

Figure 6: Execution times for random join orders of TPC-H Q5

One of the most crucial decisions a query optimizer must make is the execution order of operators, in particular the join order. Join ordering is NP-hard in general, which has caused many people to give up on it and use heuristics. But in practice even fairly complex join ordering problems can be solved exactly by using smart algorithms [23, 24]. The complexity of the problem depends upon the structure of the problem, in particular its query graph, and fortunately most users do not enter clique queries. This structure can be used directly in a graph-theoretic approach to drive the search exploration of an optimizer, either in a bottom-up [23, 24] or in a top-down [6] fashion. Again, fairly complex queries can be solved exactly, for example optimizing a chain query with 50 relations is no problem at all when using a graph-based optimization algorithm.

In the unlikely event that the user really enters a query that is too complex to solve exactly, the optimizer can degrade gracefully towards heuristics [31]: Before optimization it computes the implied size of the search space, and if that is too large it uses heuristics to rule out very unlikely join orders, until the remaining search space is tractable. This way it avoids the hard cliff of switching from an optimal solution to a greedy heuristic, and if reducing the search space required only a few heuristic pruning steps, it is very likely that the optimal solution is still in the remaining search space.

Of course all of these optimizations rely upon cardinality estimates, which are notoriously unreliable for complex queries. However, accuracy can be improved with some care. All estimates will induce errors, of course, but it is possible to derive estimates such that the impact of errors is bounded [26]. For join operators, estimation errors propagate multiplicative, but by careful histogram construction the maximum multiplicative error can be controlled; thus allowing to drive error bounds for larger join plans.

Another great problem besides error propagation is correlations. This is particularly severe for RDF data, where correlations between triples with identical subjects are not an exception but the norm. Most estimators assume more or less independence, and severely misestimate the cardinalities for SPARQL queries. However, even though RDF is conceptually very unpredicatable, real world data exhibits a certain structure. This structure can be captured by computing *characteristic sets* [36], i.e., the sets of predicates that occur together for a given subject. Conceptually this is similar to schema discovery, and identifies common structure within the data graph. By annotation these characteristic sets with statistics, the optimizer can derive very accurate estimates, even for large join groups and even in the presence of correlations. Recognizing typical query patterns is a good idea for SPARQL processing in general, as it allows both for improving the accuracy of estimates and for reducing the optimization effort [11]. Similar ideas would be useful for relational data processing, too. Star queries for example have a very large combinatorial search space, but are easy to optimize in reality. Recognizing that during optimization could greatly improve optimization times.

Finally, if estimates are really unreliable in a query, the optimizer can always execute parts of the query and optimize the rest of the query based upon the observed data [33]. This eliminates most of the uncertainty and prevents gross misestimations. This comes at a cost, these intermediate results have to be materialized. Therefore it is helpful to analyze the query and only execute those parts that are both hard to estimate and that actually have an impact on the resulting plan. Somewhat surprisingly this is not always the case. Sometimes the query structure determines parts of the execution plan, largely independent from query estimates. Sensitivity analysis helps to avoid unnecessary materialization costs.

Most of the optimizations mentioned here are useful independent from the concrete database engine, but sometimes it helps to evolve the optimizer and the runtime system simultaneously. One example for that is the *group join*, i.e., the combination of a *join* and subsequent *group by* into one algebraic operator [25]. Such constructs are useful in many aggregation queries, for example TPC-H query 13, where it improves response times significantly, but they are not always applicable. The query optimizer must therefore be extended to recognize all cases where such a compound operator is valid and useful, and the runtime system must be extended to handle all these cases. This is a trade-off, using a simpler implementation in the runtime system makes life more difficult for the query optimizer, and vice versa.

In general, query optimization is incredibly important. Changes to an already tuned runtime system might bring another 10% improvement, but changes to the query optimizer can often bring a factor 10. Or cost a factor 10, which is why some people criticize query optimization. But the impact of query optimization is so large that there are no alternatives, the runtime system alone could never get that good performance without an optimizer.

5. RELATED WORK

There are too many excellent systems out there to do them all justice. We therefore limit ourselves to just a few, and apologize for the systems that we did not mention here.

One of the most influential systems was of course System R [4], which pioneered many classical techniques like SQL, dynamic-programming based query optimization, transaction processing, locking, etc.. Its design influenced many, or perhaps even most, systems up to today. On the query processing side the Volcano project was highly influential [9]. Its query processing model, and in particular its parallelization framework, are widely used in commercial systems, Putting the ideas from these two projects together would give a rough blueprint for traditional disk-based systems. Of course architecture evolved and improved over time, but most disk-based systems were quite similar to that approach.

System architectures started to change significantly as the hardware evolved. The MonetDB project pioneered a radically different approach [12]: It used a column oriented storage layout, and optimized query processing for CPU utilization, assuming (correctly) that data could be primarily kept in main-memory on modern machines. The MonetDB/X100 project [48], that then evolved into the commercial VectorWise system [47], took this CPU orientation even further in their vectorized processing approach. Both system offer excellent OLAP performance, much better than what was possible with the older, disk-based row stores. A similar approach to MonetDB was also followed by C-Store [46], that then evolved into the commercial Vertica system, and that introduced more aggressive storage and indexing techniques.

On the OLTP side H-Store [15], and its commercial incarnation VoltDB [14], proposed a radical change from traditional architectures. Its authors observed that if data fits into main memory, a traditional database system spends most of its time during OLTP processing on “unproductive” work like latching, locking, and logging. Therefore they proposed to restrict OLTP transactions to non-interactive stored procedures, which can be executed much faster in the a serial manner, without any need for latching or locking.

Of course many people already have disk-based systems, and for them a transition to a different architecture is difficult. The Short-MT project [13] is interesting in that regard, as it took a relatively classical architecture, and successfully adapted it for multi core processing. This demonstrates that modernizing an architecture is definitively possible, but requires a lot of work. Another interesting system is the Crescando project [8], which, instead of concentrating on executing a single transaction as quickly as possible, concentrates on execution a large number of concurrent transactions as quickly as possible. At some point this batching will pay off, and outperform classical execution.

Most of the systems mentioned so far operate on one node or a small number of nodes. Another class of systems is aiming at distributed or cloud-based data processing. For example Stratosphere [3] combined query optimization techniques with distributed query processing. But this large group of systems is beyond the scope of this paper, see [44] for a survey.

The big commercial database systems have traditionally been conservative in adapting new technologies. The inner parts are changing, for example systems have spend a lot of effort on query optimization [45], but the overall architecture often still very much disk oriented. But the impact of technology change is so large that they started changing, too. SAP presented the HANA system [19], that is a columnar in-memory database system, aiming to offer both OLTP and OLAP functionality. Microsoft SQL Server has released the Hekaton module [18] that is an in-memory OLTP engine. And new IBM BLU [41] is an in-memory OLAP engine. So after a long period of architectural stability, database system start changing again, mandated by the changes to the underlying hardware.

6. CONCLUSION

Designing a high-performance database engine is a complex task. There are a few design principles that are virtually always a good idea, like accessing data sparingly, executing as few instructions as possible, and using query optimization. But we showed by the example of RDF-3X and HyPer that systems have to make trade-offs, and these trade-offs might change over time. In particular the choice between disk-based and in-memory processing has a very large impact, and nowadays the focus clearly drifts twowards in-memory processing.

But simply putting everything in main memory is not the answer, even in-memory systems have to care about locality, parallelism, interpretation overhead, and so on. The HyPer system has shown one very succesful way to design an in-memory system, but its architecture will evolve over time, too. The hardware continues to change, and so does database design.

7. REFERENCES

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[2] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[3] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130, 2010.

[4] M. W. Blasgen, M. M. Astrahan, D. D. Chamberlin, J. Gray, W. F. K. III, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, M. Schkolnick, P. G. Selinger, D. R. Slutz, H. R. Strong, I. L. Traiger, B. W. Wade, and R. A. Yost. System R: An architectural overview. *IBM Systems Journal*, 20(1):41–62, 1981.

[5] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. A. Kuno, R. O. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload ch-benchmark. In *DBTest*, page 8, 2011.

[6] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. Effective and robust pruning for top-down join enumeration algorithms. In *ICDE*, pages 414–425, 2012.

[7] F. Funke, A. Kemper, and T. Neumann. Compacting transactional data in hybrid OLTP&OLAP databases. *PVLDB*, 5(11):1424–1435, 2012.

[8] G. Giannikis, P. Unterbrunner, J. Meyer, G. Alonso, D. Fauser, and D. Kossmann. Crescando. In *SIGMOD Conference*, pages 1227–1230, 2010.

[9] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[10] A. Gubichev and T. Neumann. Path query processing on very large RDF graphs. In *WebDB*, 2011.

[11] A. Gubichev and T. Neumann. Exploiting the query structure for efficient join ordering in sparql queries. In *EDBT*, pages 439–450, 2014.

[12] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[13] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, pages 24–35, 2009.

[14] E. P. C. Jones and A. Pavlo. A specialized architecture for high-throughput OLTP applications. 13th Intl. Workshop on High Performance Transaction Systems, Oct 2009. `http://www.hpts.ws/`.

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[16] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[17] A. Kemper, T. Neumann, F. Funke, V. Leis, and H. Mühe. HyPer: Adapting columnar main-memory data management for transactional and query processing. *IEEE Data Eng. Bull.*, 35(1):46–51, 2012.

[18] P.-Å. Larson, M. Zwilling, and K. Farlee. The Hekaton memory-optimized OLTP engine. *IEEE Data Eng. Bull.*, 36(2):34–40, 2013.

[19] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krüger, and M. Grund. High-performance transaction processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[20] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*, pages 743–754, 2014.

[21] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*, pages 38–49, 2013.

[22] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, pages 580–591, 2014.

[23] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, pages 930–941, 2006.

[24] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD Conference*, pages 539–552, 2008.

[25] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *PVLDB*, 4(11):843–851, 2011.

[26] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB*, 2(1):982–993, 2009.

[27] H. Mühe, A. Kemper, and T. Neumann. Executing long-running transactions in synchronization-free main memory database systems. In *CIDR*, 2013.

[28] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. Scyper: A hybrid oltp&olap

distributed main memory database system for scalable real-time analytics. In *BTW*, pages 499–502, 2013.
[29] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *PVLDB*, 6(14):1702–1713, 2013.
[30] T. Neumann. *Efficient generation and execution of DAG-structured query graphs*. PhD thesis, Universität Mannheim, 2005.
[31] T. Neumann. Query simplification: graceful degradation for join-order optimization. In *SIGMOD Conference*, pages 403–414, 2009.
[32] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.
[33] T. Neumann and C. A. Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *BTW*, pages 73–92, 2013.
[34] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE Data Eng. Bull.*, 37(1):3–11, 2014.
[35] T. Neumann and G. Moerkotte. Generating optimal DAG-structured query evaluation plans. *Computer Science - R&D*, 24(3):103–117, 2009.
[36] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, pages 984–994, 2011.
[37] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
[38] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD Conference*, pages 627–640, 2009.
[39] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
[40] T. Neumann and G. Weikum. x-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *PVLDB*, 3(1):256–263, 2010.
[41] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.
[42] Random execution plans. `http://databasearchitects.blogspot.de/2014/06/random-execution-plans.html`, June 2014.
[43] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *ICDE*, pages 592–603, 2014.
[44] M. Saecker and V. Markl. Big data analytics on modern hardware architectures: A technology survey. In *eBISS*, pages 125–149, 2012.
[45] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.
[46] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.
[47] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.
[48] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - a DBMS in the CPU cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.