NoSQL Databases

Christof Strauch
(cs134@hdm-stuttgart.de)

Lecture
Selected Topics on Software-Technology
Ultra-Large Scale Sites

Lecturer
Prof. Walter Kriha

Course of Studies
Computer Science and Media (CSM)

University
Hochschule der Medien, Stuttgart
(Stuttgart Media University)

Contents

List of Figures

List of Tables

1. Introduction

1.1. Introduction and Overview

Relational database management systems (RDMBSs) today are the predominant technology for storing structured data in web and business applications. Since Codds paper "A relational model of data for large shared data banks" [Cod70] from 1970 these datastores relying on the relational calculus and providing comprehensive ad hoc querying facilities by SQL (cf. [CB74]) have been widely adopted and are often thought of as the only alternative for data storage accessible by multiple clients in a consistent way. Although there have been different approaches over the years such as object databases or XML stores these technologies have never gained the same adoption and market share as RDBMSs. Rather, these alternatives have either been absorbed by relational database management systems that e. g. allow to store XML and use it for purposes like text indexing or they have become niche products for e. g. OLAP or stream processing.

In the past few years, the "one size fits all"-thinking concerning datastores has been questioned by both, science and web affine companies, which has lead to the emergence of a great variety of alternative databases. The movement as well as the new datastores are commonly subsumed under the term *NoSQL*, "used to describe the increasing usage of non-relational databases among Web developers" (cf. [Oba09a]).

This paper's aims at giving a systematic overview of the motives and rationales directing this movement (chapter 2), common concepts, techniques and patterns (chapter 3) as well as several classes of NoSQL databases (key-/value-stores, document databases, column-oriented databases) and individual products (chapters 4–6).

1.2. Uncovered Topics

This paper excludes the discussion of datastores existing before and are not referred to as part of the NoSQL movement, such as object-databases, pure XML databases and DBMSs for special purposes (such as analytics or stream processing). The class of graph databases is also left out of this paper but some resources are provided in the appendix A. It is out of the scope of this paper to suggest individual NoSQL datastores in general as this would be a total misunderstanding of both, the movement and the approach of the NoSQL datastores, as not being "one size fits all"-solutions. In-depth comparisons between all available NoSQL databases would also exceed the scope of this paper.

2. The NoSQL-Movement

In this chapter, motives and main drivers of the NoSQL movement will be discussed along with remarks passed by critics and reactions from NoSQL advocates. The chapter will conclude by different attempts to classify and characterize NoSQL databases. One of them will be treated in the subsequent chapters.

2.1. Motives and Main Drivers

The term NoSQL was first used in 1998 for a relational database that omitted the use of SQL (see [Str10]). The term was picked up again in 2009 and used for conferences of advocates of non-relational databases such as Last.fm developer Jon Oskarsson, who organized the NoSQL meetup in San Francisco (cf. [Eva09a]). A blogger, often referred to as having made the term popular is Rackspace employee Eric Evans who later described the ambition of the NoSQL movement as "the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for" (cf. [Eva09b]).

This section will discuss rationales of practitioners for developing and using nonrelational databases and display theoretical work in this field. Furthermore, it will treat the origins and main drivers of the NoSQL movement.

2.1.1. Motives of NoSQL practioners

The Computerworld magazine reports in an article about the NoSQL meet-up in San Francisco that "NoSQLers came to share how they had overthrown the tyranny of slow, expensive relational databases in favor of more efficient and cheaper ways of managing data." (cf. [Com09a]). It states that especially Web 2.0 startups have begun their business without Oracle and even without MySQL which formerly was popular among startups. Instead, they built their own datastores influenced by Amazon's Dynamo ([DHJ+07]) and Google's Bigtable ([CDG+06]) in order to store and process huge amounts of data like they appear e. g. in social community or cloud computing applications; meanwhile, most of these datastores became open source software. For example, Cassandra originally developed for a new search feature by Facebook is now part of the Apache Software Project. According to engineer Avinash Lakshman, it is able to write 2500 times faster into a 50 gigabytes large database than MySQL (cf. [LM09]).

The Computerworld article summarizes reasons commonly given to develop and use NoSQL datastores:

Avoidance of Unneeded Complexity Relational databases provide a variety of features and strict data consistency. But this rich feature set and the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases.

As an example, Adobe's ConnectNow holds three copies of user session data; these replicas do not neither have to undergo all consistency checks of a relational database management systems nor do they have to be persisted. Hence, it is fully sufficient to hold them in memory (cf. [Com09b]).

High Throughput Some NoSQL databases provide a significantly higher data throughput than traditional RDBMSs. For instance, the column-store Hypertable which pursues Google's Bigtable approach allows the local search engine Zvent to store one billion data cells per day [Jud09]. To give another example, Google is able to process 20 petabyte a day stored in Bigtable via it's MapReduce approach [Com09b].

Horizontal Scalability and Running on Commodity Hardware "Definitely, the volume of data is getting so huge that people are looking at other technologies", says Jon Travis, an engineer at SpringSource (cited in [Com09a]). Blogger Jonathan Ellis agrees with this notion by mentioning three problem areas of current relational databases that NoSQL databases are trying to address (cf. [Ell09a]):

1. Scale out data (e.g. 3 TB for the *green badges* feature at Digg, 50 GB for the inbox search at Facebook or 2 PB in total at eBay)
2. Performance of single servers
3. Rigid schema design

In contrast to relational database management systems most NoSQL databases are designed to scale well in the horizontal direction and not rely on highly available hardware. Machines can be added and removed (or crash) without causing the same operational efforts to perform sharding in RDBMS cluster-solutions; some NoSQL datastores even provide automatic sharding (such as MongoDB as of March 2010, cf. [Mer10g]). Javier Soltero, CTO of SpringSource puts it this way: "Oracle would tell you that with the right degree of hardware and the right configuration of Oracle RAC (Real Application Clusters) and other associated magic software, you can achieve the same scalability. But at what cost?" (cited in [Com09a]). Especially for Web 2.0 companies the scalability aspect is considered crucial for their business, as Johan Oskarsson of Last.fm states: "Web 2.0 companies can take chances and they need scalability. When you have these two things in combination, it makes [NoSQL] very compelling." (Johan Oskarsson, Organizer of the meet-up and web developer at Last.fm, cf. [Com09a]). Blogger Nati Shalom agrees with that: "cost pressure also forced many organizations to look at more cost-effective alternatives, and with that came research that showed that distributed storage based on commodity hardware can be even more reliable then[sic!] many of the existing high end databases" (cf. [Sha09a] and for further reading [Sha09c]). He concludes: "All of this led to a demand for a cost effective "scale-first database"".

Avoidance of Expensive Object-Relational Mapping Most of the NoSQL databases are designed to store data structures that are either simple or more similar to the ones of object-oriented programming languages compared to relational data structures. They do not make expensive object-relational mapping necessary (such as Key/Value-Stores or Document-Stores). This is particularly important for applications with data structures of low complexity that can hardly benefit from the features of a relational database. Dare Obasanjo claims a little provokingly that "all you really need [as a web developer] is a key<->value or tuple store that supports some level of query functionality and has decent persistence semantics." (cf. [Oba09a]). The blogger and database-analyst Curt Monash iterates on this aspect: "SQL is an awkward fit for procedural code, and almost all code is procedural. [For data upon which users expect to do heavy, repeated manipulations, the cost of mapping data into SQL is] well worth paying [...] But when your database structure is very, very simple, SQL may not seem that beneficial." Jon Travis, an engineer at SpringSource agrees with that: "Relational databases give you too much. They force you to twist your object data to fit a RDBMS." (cited in [Com09a]).

In a blog post on the Computerworld article Nati Shalom, CTO and founder of GigaSpaces, identifies the following further drivers of the NoSQL movement (cf. [Sha09b]):

Complexity and Cost of Setting up Database Clusters He states that NoSQL databases are designed in a way that "PC clusters can be easily and cheaply expanded without the complexity and cost of 'sharding,' which involves cutting up databases into multiple tables to run on large clusters or grids".

Compromising Reliability for Better Performance Shalom argues that there are "different scenarios where applications would be willing to compromise reliability for better performance." As an example of such a scenario favoring performance over reliability, he mentions HTTP session data which "needs to be shared between various web servers but since the data is transient in nature (it goes away when the user logs off) there is no need to store it in persistent storage."

The Current "One size fit's it all" Databases Thinking Was and Is Wrong Shalom states that "a growing number of application scenarios cannot be addressed with a traditional database approach". He argues that "this realization is actually not that new" as the studies of Michael Stonebraker (see below) have been around for years but the old 'news' has spread to a larger community in the last years. Shalom thinks that this realization and the search for alternatives towards traditional RDBMSs can be explained by two major trends:

1. The continuous growth of data volumes (to be stored)
2. The growing need to process larger amounts of data in shorter time

Some companies, especially web-affine ones have already adopted NoSQL databases and Shalom expects that they will find their way into mainstream development as these datastores mature. Blogger Dennis Forbes agrees with this issue by underlining that the requirements of a bank are not universal and especially social media sites have different characteristics: "unrelated islands of data", a "very low [...] user/transaction value" and no strong need for data integrity. Considering these characteristics he states the following with regard to social media sites and big web applications:

> "The truth is that you don't need ACID for Facebook status updates or tweets or Slashdots comments. So long as your business and presentation layers can robustly deal with inconsistent data, it doesn't really matter. It isn't ideal, obviously, and preferrably [sic!] you see zero data loss, inconsistency, or service interruption, however accepting data loss or inconsistency (even just temporary) as a possibility, breaking free of by far the biggest scaling "hindrance" of the RDBMS world, can yield dramatic flexibility. [...]
>
> This is the case for many social media sites: data integrity is largely optional, and the expense to guarantee it is an unnecessary expenditure. When you yield pennies for ad clicks after thousands of users and hundreds of thousands of transactions, you start to look to optimize." (cf. [For10])

Shalom suggests caution when moving towards NoSQL solutions and to get familiar with their specific strengths and weaknesses (e. g. the ability of the business logic to deal with inconsistency). Others, like David Merriman of 10gen (the company behind MongoDB) also stress that there is no single tool or technology for the purpose of data storage but that there is a segmentation currently underway in the database field bringing forth new and different data stores for e. g. business intelligence vs. online transaction processing vs. persisting large amounts of binary data (cf. [Tec09]).

The Myth of Effortless Distribution and Partitioning of Centralized Data Models Shalom further addresses the myth surrounding the perception that data models originally designed with a single database in mind (centralized datamodels, as he puts it) often cannot easily be partitioned and distributed among database servers. This signifies that without further effort, the application will neither necessarily scale and nor work correct any longer. The professionals of Ajatus agree with this in a blog post stating that if a database grows, at first, replication is configured. In addition, as the amount of data grows further, the database is sharded by expensive system admins requiring large financial sums or a fortune worth of money for commercial DBMS-vendors are needed to operate the sharded database (cf. [Aja09]). Shalom reports from an architecture summit at eBay in the summer of 2009. Participants agreed on the fact that although typically, abstractions involved trying to hide distribution and partitioning issues away from applications (e. g. by proxy layers routing requests to certain servers) "this abstraction cannot insulate the application from the reality that [. . .] partitioning and distribution is involved. The spectrum of failures within a network is entirely different from failures within a single machine. The application needs to be made aware of latency, distributed failures, etc., so that it has enough information to make the correct context-specific decision about what to do. The fact that the system is distributed leaks through the abstraction." ([Sha09b]). Therefore he suggests designing datamodels to fit into a partioned environment even if there will be only one centralized database server initially. This approach offers the advantage to avoid exceedingly late and expensive changes of application code.

Shalom concludes that in his opinion relational database management systems will not disappear soon. However, there is definitely a place for more specialized solutions as a "one size fits all" thinking was and is wrong with regards to databases.

Movements in Programming Languages and Development Frameworks The blogger David Intersimone additionally observes movements in programming languages and development frameworks that provide abstractions for database access trying to hide the use of SQL (cf. []) and relational databases ([Int10]). Examples for this trend in the last couple of years include:

- Object-relational mappers in the Java and .NET world like the Java Persistence API (JPA, part of the EJB 3 specification, cf. [DKE06], [BO06],), implemented by e. g. Hibernate ([JBo10a], or the LINQ-Framework (cf. [BH05]) with its code generator SQLMetal and the ADO.NET Entity Framework (cf. [Mic10]) since .NET version 4.
- Likewise, the popular Ruby on Rails (RoR, [HR10]) framework and others try to hide away the usage of a relational database (e. g. by implementing the active record pattern as of RoR).
- NoSQL datastores as well as some databases offered by cloud computing providers completely omit a relational database. One example of such a cloud datastore is Amazon's SimpleDB, a schema-free, Erlang-based eventually consistent datastore which is characterized as an Entity-Attribute-Value (EAV). It can store large collections of items which themselves are hashtables containing attributes that consist of key-value-pairs (cf. [Nor09]).

The NoSQL databases react on this trend and try to provide data structures in their APIs that are closer to the ones of programming languages (e. g. key/value-structures, documents, graphs).

Requirements of Cloud Computing In an interview Dwight Merriman of 10gen (the company behind MongoDB) mentions two major requirements of datastores in cloud computing environments ([Tec09]):

1. High until almost ultimate scalability—especially in the horizontal direction
2. Low administration overhead

In his view, the following classes of databases work well in the cloud:

- Data warehousing specific databases for batch data processing and map/reduce operations.
- Simple, scalable and fast key/value-stores.
- Databases containing a richer feature set than key/value-stores fitting the gap with traditional RDBMSs while offering good performance and scalability properties (such as document databases).

Blogger Nati Shalom agrees with Merriman in the fact that application areas like cloud-computing boosted NoSQL databases: "what used to be a niche problem that only a few fairly high-end organizations faced, became much more common with the introduction of social networking and cloud computing" (cf. [Sha09a]).

The RDBMS plus Caching-Layer Pattern/Workaround vs. Systems Built from Scratch with Scalability in Mind In his article "MySQL and memcached: End of an era?" Todd Hoff states that in a "pre-cloud, relational database dominated world" scalability was an issue of "leveraging MySQL and memcached":

> "Shard MySQL to handle high write loads, cache objects in memcached to handle high read loads, and then write a lot of glue code to make it all work together. That was state of the art, that was how it was done. The architecture of many major sites still follow[sic!] this pattern today, largely because with enough elbow grease, it works." (cf. [Hof10c])

But as scalability requirements grow and these technologies are less and less capable to suit with them. In addition, as NoSQL datastores are arising Hoff comes to the conclusion that "[with] a little perspective, it's clear the MySQL + memcached era is passing. It will stick around for a while. Old technologies seldom fade away completely." (cf. [Hof10c]). As examples, he cites big websites and players that have moved towards non-relational datastores including LinkedIn, Amazon, Digg and Twitter. Hoff mentions the following reasons for using NoSQL solutions which have been explained earlier in this paper:

- Relational databases place computation on reads, which is considered wrong for large-scale web applications such as Digg. NoSQL databases therefore do not offer or avoid complex read operations.
- The serial nature of applications[1] often waiting for I/O from the data store which does no good to scalability and low response times.
- Huge amounts of data and a high growth factor lead Twitter towards facilitating Cassandra, which is designed to operate with large scale data.
- Furthermore, operational costs of running and maintaining systems like Twitter escalate. Web applications of this size therefore "need a system that can grow in a more automated fashion and be highly available." (cited in [Hof10c]).

For these reasons and the "clunkiness" of the MySQL and memcached era (as Hoff calls it) large scale (web) applications nowadays can utilize systems built from scratch with scalability, non-blocking and asynchronous database I/O, handling of huge amounts of data and automation of maintainance and operational tasks in mind. He regards these systems to be far better alternatives compared to relational DBMSs with additional object-caching. Amazon's James Hamilton agrees with this by stating that for many large-scale web sites scalability from scratch is crucial and even outweighs the lack of features compared to traditional RDBMSs:

[1] Hoff does not give more detail on the types of applications meant here, but—in the authors opinion—a synchronous mindset and implementation of database I/O can be seen in database connectivity APIs (such as ODBC or JDBC) as well as in object-relational mappers and it spreads into many applications from these base technologies.

> "Scale-first applications are those that absolutely must scale without bound and being able to do this without restriction is much more important than more features. These applications are exemplified by very high scale web sites such as Facebook, MySpace, Gmail, Yahoo, and Amazon.com. Some of these sites actually do make use of relational databases but many do not. The common theme across all of these services is that scale is more important than features and none of them could possibly run on a single RDBMS." (cf. [Ham09] cited in [Sha09a])

Yesterday's vs. Today's Needs In a discussion on CouchDB Lehnardt and Lang point out that needs regarding data storage have considerably changed over time (cf. [PLL09]; this argument is iterated further by Stonebraker, see below). In the 1960s and 1970s databases have been designed for single, large high-end machines. In contrast to this, today, many large (web) companies use commodity hardware which will predictably fail. Applications are consequently designed to handle such failures which are considered the "standard mode of operation", as Amazon refers to it (cf. [DHJ+07, p. 205]). Furthermore, relational databases fit well for data that is rigidly structured with relations and allows for dynamic queries expressed in a sophisticated language. Lehnhardt and Lang point out that today, particularly in the web sector, data is neither rigidly structured nor are dynamic quieries needed as most applications already use prepared statements or stored procedures. Therefore, it is sufficient to predefine queries within the database and assign values to their variables dynamically (cf. [PLL09]).

Furthermore, relational databases were initially designed for centralized deployments and not for distribution. Although enhancements for clustering have been added on top of them it still leaks through that traditional were not designed having distribution concepts in mind at the beginning (like the issues adverted by the "fallaclies of network computing" quoted below). As an example, synchronization is often not implemented efficiently but requires expensive protocols like two or three phase commit. Another difficulty Lehnhardt and Lang see is that clusters of relational databases try to be "transparent" towards applications. This means that the application should not contain any notion if talking to a singly machine or a cluster since all distribution aspects are tried to be hidden from the application. They question this approach to keep the application unaware of all consequences of distribution that are e. g. stated in the famous eight fallacies of distributed computing (cf. [Gos07][2]:

> "Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.
>
> 1. The network is reliable
> 2. Latency is zero
> 3. Bandwidth is infinite
> 4. The network is secure
> 5. Topology doesn't change
> 6. There is one administrator
> 7. Transport cost is zero

[2] The fallacies are cited according to James Gosling's here. There is some discussion about who came up with the list: the first seven fallacies are commonly credited to Peter Deutsch, a Sun Fellow, having published them in 1994. Fallacy number eight was added by James Gosling around 1997. Though—according to the English Wikipedia—"Bill Joy and Tom Lyon had already identified the first four as "The Fallacies of Networked Computing"" (cf. [Wik10]). More details on the eight fallacies can be found in an article of Rotem-Gal-Oz (cf. [RGO06])

8. The network is homogeneous"

While typical business application using relational database management systems try, in most cases, to hide distribution aspects from the application (e. g. by clusters, persistence layers doing object-relational mapping) many large web companies as well as most of the NoSQL databases do not pursue this approach. Instead, they let the application know and leverage them. This is considered a paradigm shift in the eyes of Lehnardt and Lang (cf. [PLL09]).

Further Motives In addition to the aspects mentioned above David Intersimone sees the following three goals of the NoSQL movement (cf. [Int10]):

- Reach less overhead and memory-footprint of relational databases
- Usage of Web technologies and RPC calls for access
- Optional forms of data query

2.1.2. Theoretical work

In their widely adopted paper "The End of an Architectural Era" (cf. [SMA+07]) Michael Stonebraker[3] et al. come to the conclusion "that the current RDBMS code lines, while attempting to be a "one size fits all" solution, in fact, excel at nothing". *Nothing* in this context means that they can neither compete with "specialized engines in the data warehouse, stream processing, text, and scientific database markets" which outperform them "by 1–2 orders of magnitude" (as shown in previous papers, cf. [Sc05], [SBc+07]) nor do they perform well in their their home market of business data processing / online transaction processing (OLTP), where a prototype named H-Store developed at the M.I.T. beats up RDBMSs by nearly two orders of magnitude in the TPC-C benchmark. Because of these results they conclude that RDBMSs" are 25 year old legacy code lines that should be retired in favor of a collection of "from scratch" specialized engines. The DBMS vendors (and the research community) should start with a clean sheet of paper and design systems for tomorrow's requirements, not continue to push code lines and architectures designed for yesterday's needs". But how do Stonebraker et al. come to this conclusion? Which inherent flaws do they find in relational database management systems and which suggestions do they provide for the "complete rewrite" they are requiring?

At first, Stonebraker et al. argue that RDBMSs have been architected more than 25 years ago when the hardware characteristics, user requirements and database markets where different from those today. They point out that "popular relational DBMSs all trace their roots to System R from the 1970s": IBM's DB2 is a direct descendant of System R, Microsoft's SQL Server has evolved from Sybase System 5 (another direct System R descendant) and Oracle implemented System R's user interface in its first release. Now, the architechture of System R has been influenced by the hardware characteristics of the 1970s. Since then, processor speed, memory and disk sizes have increased enormously and today do not limit programs in the way they did formerly. However, the bandwidth between hard disks and memory has not increased as fast as the CPU speed, memory and disk size. Stonebraker et al. criticize that this development in the field of hardware has not impacted the architecture of relational DBMSs. They especially see the following architectural characteristics of System R shine through in today's RDBMSs:

- "Disk oriented storage and indexing structures"
- "Multithreading to hide latency"

[3] When reading and evaluating Stonebraker's writings it has to be in mind that he is commercially involved into multiple DBMS products such Vertica, a column-store providing data warehousing and business analytics.

- "Locking-based concurrency control mechanisms"
- "Log-based recovery"

In this regard, they underline that although "there have been some extensions over the years, including support for compression, shared-disk architectures, bitmap indexes, support for user-defined data types and operators [...] no system has had a complete redesign since its inception".

Secondly, Stonebraker et al. point out that new markets and use cases have evolved since the 1970s when there was only business data processing. Examples of these new markets include "data warehouses, text management, and stream processing" which "have very different requirements than business data processing". In a previous paper (cf. [Sc05]) they have shown that RDBMSs "could be beaten by specialized architectures by an order of magnitude or more in several application areas, including:

- Text (specialized engines from Google, Yahoo, etc.)
- Data Warehouses (column stores such as Vertica, Monet, etc.)
- Stream Processing (stream processing engines such as StreamBase and Coral8)
- Scientific and intelligence databases (array storage engines such as MATLAB and ASAP)"

They go on noticing that user interfaces and usage model also changed over the past decades from terminals where "operators [were] inputting queries" to rich client and web applications today where interactive transactions and direct SQL interfaces are rare.

Stonebraker et al. now present "evidence that the current architecture of RDBMSs is not even appropriate for business data processing". They have designed a DBMS engine for OLTP called H-Store that is functionally equipped to run the TPC-C benchmark and does so 82 times faster than a popular commercial DBMS. Based on this evidence, they conclude that "there is no market where they are competitive. As such, they should be considered as legacy technology more than a quarter of a century in age, for which a complete redesign and re-architecting is the appropriate next step".

Design Considerations

In a section about design considerations Stonebraker et al. explain why relational DBMSs can be outperformed even in their home market of business data processing and how their own DMBS prototype H-Store can "achieve dramatically better performance than current RDBMSs". Their considerations especially reflect the hardware development over the past decades and how it could or should have changed the architecture of RDBMSs in order to gain benefit from faster and bigger hardware, which also gives hints for "the complete rewrite" they insist on.

Stonebraker et al. see five particularly significant areas in database design:

Main Memory As—compared to the 1970s—enormous amounts of main memory have become cheap and available and as "The overwhelming majority of OLTP databases are less than 1 Tbyte in size and growing [...] quite slowly" they conclude that such databases are "capable of main memory deployment now or in near future". Stonebraker et al. therefore consider the OLTP market a main memory market even today or in near future. They criticize therefore that "the current RDBMS vendors have disk-oriented solutions for a main memory problem. In summary, 30 years of Moore's law has antiquated the disk-oriented relational architecture for OLTP applications". Although there are relational databases operating in memory (e. g. TimesTen, SolidDB) these systems also inherit "baggage"—as Stonebraker et al. call it—from System R, e. g. disk-based recovery logs or dynamic locking, which have a negative impact on the performance of these systems.

Multi-Threading and Resource Control As discussed before, Stonebraker et al. consider databases a main-memory market. They now argue that transactions typically affect only a few data sets that have to be read and/or written (at most 200 read in the TPC-C benchmark for example) which is very cheap if all of this data is kept in memory and no disk I/O or user stalls are present. As a consequence, the do not see any need for multithreaded execution models in such main-memory databases which makes a considerable amount of "elaborate code" of conventional relational databases irrelevant, namely multi-threading systems to maximize CPU- and disk-usage, resource governors limiting load to avoid resource exhausting and multi-threaded datastructures like concurrent B-trees. "This results in a more reliable system, and one with higher performance", they argue. To avoid long running transactions in such a single-threaded system they require either the application to break up such transactions into smaller ones or—in the case of analytical purposes—to run these transactions in data warehouses optimized for this job.

Grid Computing and Fork-Lift Upgrades Stonebraker et al. furthermore outline the development from shared-memory architectures of the 1970s over shared-disk architectures of the 1980s towards shared-nothing approaches of today and the near future, which are "often called grid computing or blade computing". As a consequence, Stonebraker et al. insist that databases have to reflect this development, for example (and most obviously) by horizontal partitioning of data over several nodes of a DBMS grid. Furthermore, they advocate for incremental horizontal expansion of such grids without the need to reload any or all data by an administrator and also without downtimes. They point out that these requirements have significant impact on the architecture of DBMSs—e. g. the ability to transfer parts of the data between nodes without impacting running transactions—which probably cannot be easily added to existing RDBMSs but can be considered in the design of new systems (as younger databases like Vertica show).

High Availability The next topics Stonebraker et al. address are high availability and failover. Again, they outline the historical development of these issues from log-tapes that have been sent off site by organizations and were run on newly delivered hardware in the case of disaster over disaster recovery services installing log-tapes on remote hardware towards hot standby or multiple-site solutions that are common today. Stonebraker et al. regard high availability and built-in disaster recovery as a crucial feature for DBMSs which—like the other design issues they mention—has to be considered in the architecture and design of these systems. They particularly require DBMSs in the OLTP field to

1. "keep multiple replicas consistent, requiring the ability to run seamlessly on a grid of geographically dispersed systems"
2. "start with shared-nothing support at the bottom of the system" instead of gluing "multi-machine support onto [...] SMP architectures."
3. support a shared-nothing architecture in the best way by using "multiple machines in a peer-to-peer configuration" so that "load can be dispersed across multiple machines, and inter-machine replication can be utilized for fault tolerance". In such a configuration all machine resources can be utilized during normal operation and failures only cause a degraded operation as fewer resources are available. In contrast, todays HA solutions having a hot standby only utilize part of the hardware resources in normal operation as standby machines only wait for the live machines to go down. The conclude that "[these] points argue for a complete redesign of RDBMS engines so they can implement peer-to-peer HA in the guts of a new architecture" they conclude this aspect.

In such a highly available system that Stonebraker et al. require they do not see any need for a redo log as in the case of failure a dead site resuming activity "can be refreshed from the data on an operational site". Thus, there is only a need for an undo log allowing to rollback transactions. Such an undo log does not have to be persisted beyond a transaction and therefore "can be a main memory data structure that

is discarded on transaction commit". As "In an HA world, one is led to having no persistent redo log, just a transient undo one" Stonebraker et al. see another potential to remove complex code that is needed for recovery from a redo log; but they also admit that the recovery logic only changes "to new functionality to bring failed sites up to date from operational sites when they resume operation".

No Knobs Finally, Stonebraker et al. point out that current RDBMSs were designed in an "era, [when] computers were expensive and people were cheap. Today we have the reverse. Personnel costs are the dominant expense in an IT shop". They especially criticize that "RDMBSs have a vast array of complex tuning knobs, which are legacy features from a bygone era" but still used as automatic tuning aids of RDBMSs "do not produce systems with anywhere near the performance that a skilled DBA can produce". Instead of providing such features that only try to figure out a better configuration for a number of knobs Stonebraker et al. require a database to have no such knobs at all but to be ""self-everything" (self-healing, self-maintaining, self- tuning, etc.)".

Considerations Concerning Transactions, Processing and Environment

Having discussed the historical development of the IT business since the 1970s when RDBMSs were designed and the consequences this development should have had on their architecture Stonebraker et al. now turn towards other issues that impact the performance of these systems negatively:

- Persistent redo-logs have to be avoided since they are "almost guaranteed to be a significant performance bottleneck". In the HA/failover system discussed above they can be omitted totally.
- Communication between client and DBMS-server via JDBC/ODBC-like interfaces is the next performance-degrading issue they address. Instead of such an interface they "advocate running application logic – in the form of stored procedures – "in process" inside the database system" to avoid "the inter-process overheads implied by the traditional database client / server model."
- They suggest furthermore to eliminate an undo-log "wherever practical, since it will also be a significant bottleneck".
- The next performance bottleneck addressed is dynamic locking to allow concurrent access. The cost of dynamic locking should also be reduced or eliminated.
- Multi-threaded datastructures lead to latching of transactions. If transaction runtimes are short, a single-threaded execution model can eliminate this latching and the overhead associated with multi-threaded data structures "at little loss in performance".
- Finally, two-phase-commit (2PC) transactions should be avoided whenever possible as the network round trips caused by this protocol degrade performance since they "often take the order of milliseconds".

If these suggestions can be pursued depends on characteristics of OLTP transactions and schemes as Stonbraker et al. point out subsequently.

Transaction and Schema Characteristics

In addition to hardware characteristics, threading model, distribution or availability requirements discussed above, Stonebraker et al. also point out that the characteristics of database schemes as well as transaction properties also significantly influence the performance of a DBMS. Regarding database schemes and transactions they state that the following characteristics should be exploited by a DBMS:

Tree Schemes are database schemes in which "every table except a single one called *root*, has exactly one join term which is a 1-n relationship with its ancestor. Hence, the schema is a tree of 1-n relationships". Schemes with this property are especially easy to distribute among nodes of a grid "such that all equi-joins in the tree span only a single site". The root table of such a schema may be typically partitioned by it's primary key and moved to the nodes of a grid, so that on each node has the partition of the root table together with the data of all other tables referencing the primary keys in that root table partition.

Constrained Tree Application (CTA) in the notion of Stonebraker et al. is an applications that has a tree schema and only runs transactions with the following characteristics:

1. "every command in every transaction class has equality predicates on the primary key(s) of the root node"
2. "every SQL command in every transaction class is local to one site"

Transaction classes are collections of "the same SQL statements and program logic, differing in the run-time constants used by individual transactions" which Stonebraker et al. require to be defined in advance in their prototype H-Store. They furthermore argue that current OLTP applications are often designed to be CTAs or that it is at least possible to decompose them in that way and suggest schema transformations to be applied systematically in order to make an application CTA (cf. [SMA+07, page 1153]. The profit of these efforts is that "CTAs [...] can be executed very efficiently".

Single-Sited Transactions can be executed to completion on only one node without having to communicate with other sites of a DBMS grid. Constrained tree application e. g. have that property.

One-Shot Applications entirely consist of "transactions that can be executed in parallel without requiring intermediate results to be communicated among sites". In addition, queries in one-shot applications never use results of earlier queries. These properties allow the DBMS to decompose transactions "into a collection of single-site plans which can be dispatched to the appropriate sites for execution". A common technique to make applications one-shot is to partition tables vertically among sites.

Two-Phase Transactions are transactions that contain a first phase of read operations, which can—depending on its results—lead to an abortion of the transaction, and a second phase of write operations which are guaranteed to cause no integrity violations. Stonebraker et al. argue that a lot of OLTP transactions have that property and therefore exploit it in their H-Store prototype to get rid of the undo-log.

Strongly Two-Phase Transactions in addition to two-phase transactions have the property that in the second phase all sites either rollback or complete the transaction.

Transaction Commutativity is defined by Stonebraker et al. as follows: "Two concurrent transactions from the same or different classes *commute* when any interleaving of their single-site sub-plans produces the same final database state as any other interleaving (assuming both transactions commit)".

Sterile Transactions Classes are those that commute "with all transaction classes (including itself)".

H-Store Overview

Having discussed the parameters that should be considered when designing a database management system today, Stonebraker et al. sketch their protoype H-Store that performs significantly better than a commercial DBMS in the TPC-C benchmark. Their system sketch shall not need to be repeated in this paper (details can be found in [SMA+07, p. 154ff], but a few properties shall be mentioned:

- H-Store runs on a grid

- On each rows of tables are placed contiguously in main memory
- B-tree indexing is used
- Sites are partitioned into logical sites which are dedicated to one CPU core
- Logical sites are completely independent having their own indexes, tuple storage and partition of main memory of the machine they run on
- H-Store works single-threaded and runs transactions uninterrupted
- H-Store allows to run only predefined transactions implemented as stored procedures
- H-Store omits the redo-log and tries to avoid writing an undo-log whenever possible; if an undo-log cannot be avoided it is discarded on transaction commit
- If possible, query execution plans exploit the single-sited and one-shot properties discussed above
- H-Store tries to achieve the no-knobs and high availability requirements as well as transformation of transactions to be single-sited by "an automatical physical database designer which will specify horizontal partitioning, replication locations, indexed fields"
- Since H-Store keeps replicas of each table these have to be updated transactionally. Read commands can go to any copy of a table while updates are directed to all replicas.
- H-Store leverages the above mentioned schema and transaction characteristics for optimizations, e. g. omitting the undo-log in two-phase transactions.

TPC-C Benchmark

When comparing their H-Store prototype with a commercial relational DBMS Stonebraker et al. apply some important tricks to the benchmarks implementation. First, they partition the database scheme and replicate parts of it in such a way that "the schema is decomposed such that each site has a subset of the records rooted at a distinct partition of the warehouses". Secondly, they discuss how to profit of transaction characteristics discussed above. If the benchmark would run "on a single core, single CPU machine" then "every transaction class would be single-sited, and each transaction can be run to completion in a single-threaded environment". In a "paired-HA site [...]all transaction classes can be made strongly two-phase, meaning that all transactions will either succeed or abort at both sites. Hence, on a single site with a paired HA site, ACID properties are achieved with no overhead whatsoever." By applying some further tricks, they achieve that "with the basic strategy [of schema partitioning and replication (the author of this paper)] augmented with the tricks described above, all transaction classes become one-shot and strongly two-phase. As long as we add a short delay [...], ACID properties are achieved with no concurrency control overhead whatsoever."

Based on this setting, they achieved a 82 times better performance compared to a commercial DBMS. They also analyzed the overhead of performance in the commercial DBMS and examined that it was mainly caused by logging and concurrency control.

It has to be said that Stonebraker et al. implemented only part of the TPC-C benchmark and do not seem to have adapted the TPC-C benchmark to perfectly fit with the commercial DBMS as they did for their own H-Store prototype although they hired a professional DBA to tune the DBMS they compared to H-Store and also tried to optimize the logging of this system to allow it to perform better.