There's a not so new kid in town

For decades, chosing a classical database system for application data has been a fairly safe bet. That, however, changed in the late 00's…

Classical database systems, whether being from one of the major vendors (SQL Server, Oracle, DB2) or an open source alternative (MySQL, PostgreSQL) all share the same characteristics. This document briefly describes these commonalities and outlines alternative solutions that may be valuable to consider.

Once upon a time…
The 70's was a significant decade in many ways. Besides an oil crisis, Watergate, The Jackson 5 and Saturday Night Fever, it was also the decade where Ted Codd proposed the *relational model* for databases. Early systems, Ingres and System R, also arrived with an implementation of this model. They were based on architectural principles that descendants, such as SQL Server, Oracle, etc., adopted to a large extend, and kept more or less unaltered for over 30 years.

General-purpose row stores
The principles were introduced when machine power was expensive, manpower cheap and transactions were often long-runnng. They consist of a few key elemens: row-wise storing of data, locking mechanisms, buffer pool management, logging and handling of multi-threading. The classical systems can be referred to as *general-purpose traditional row stores* (GPTRS), because they really are general purpose in the sense *one size fits all*. This means, however, that they do not really excel in any specific areas either. In fact, the overhead induced by all the above mechanisms constitutes a whole 87% of the clock cycles used, leaving only 13% to the actual work!

GPTRS systems also fall short when it comes to scaling out to multiple machines. Actually, they are not built for scaling out at all, and scaling must therefore be done outside the core database.

Things have changed
Before the 00's, the alternatives to GPTRS were limited, and it was therefore a relatively safe bet to go with a GPTRS when designing a new application. But in 2006, Google published a paper about their *Bigtable* storage, which revealed how they handled the massive scale of their data by storing it in, what can be considered as a gigant distributed hash table. It was a taste of the *NoSQL* paradigm which exploded in popularity shortly thereafter. Let's zoom in a bit…

A common property of the new datastores is that they are built for scaling out to a large number of machines, transparenty without the need for external sharding logic. Often, adding new machines to the cluster is very easy and data may automatically migrate to the new nodes when they arrive.

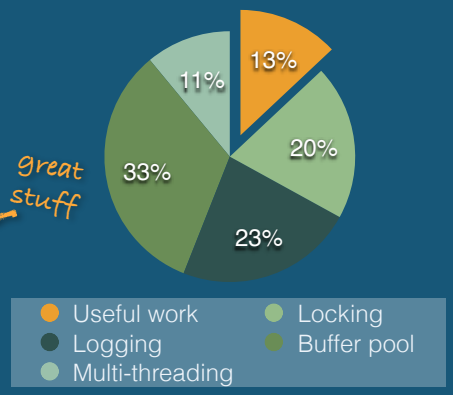
Many datastores handle replication to multiple nodes out-of-the-box, even across data centers, thereby reducing the risk of data loss.

Some datastores are schema-less, allowing for dynamic upgrades of the data model without the need for long-running table alterings.

Graph databases specializes in associative data sets by storing nodes and edges explicitly.

And for analytical processing, dedicated column stores allow for very efficient column-wise data compression with a significant gain in throughput as a consequence.

So…
There is nowadays a large variety of specialized datastores offering real scale-out, replication and high throughput out-of-the-box. Cases exist where a GPTRS is still the right choice, but there *can* be a huge potential in knowing the alternatives before deciding on a datastore.



Recommended reading

- Ten Rules for Scalable Perfomance in "Simple Operation" Datastores by Michael Stonebraker and Rick Cattell
- Cassandra (extensible record store)
- MongoDB (document store)
- Redis (in-memory key-value store)
- Neo4j (graph database)
- Vertica (column store)
- VoltDB (in-memory SQL)

Rasmus Resen Amossen, June 2015