Working Group Report on Database Engine
Chair: Daniel Abadi and Thomas Neumann

1. Introduction

This document contains the pre-meeting working group report on the database engine. We discuss important research problems in this area, what has already been accomplished, and what are important parts of these problems that we need to work on now. We do not expect to have identified all the important problems at this stage of the report generation process. This report will continue to get updated as the meeting progresses.

The working group members should feel free to edit the text in this report directly. If you delete text, please don't remove it entirely. Instead, use strike-through font so deleted text is easy to identify without tracking changes from prior versions. If you add text, please use a different color than black, so new text is easy to identify. Ideally, you should leave a comment as well, identifying yourself as the person behind the addition/subtraction, and any other justification (if you feel a justification is necessary).

The report is divided into three sections, and these sections are divided into subsections corresponding to important subtopics of the database engine that came up in the pre-meeting survey: (1) Query engine (2) Transaction Processing (3) Data lakes (4) Interoperability across multiple data systems and different type systems (engine aspects) (5) ML for Database Internals (6) Impact of hardware trends (7) Open-source.

The first of these three sections discuss some important problems in each of these subtopics, and our accomplishments so far in solving these problems. The next section discusses what we, as a database community, should be doing to address these problems, and some of our research goals in this area. In the final section we discuss our relationship to other research communities.

2. Research problems and our accomplishments thus far as a community

Query engine

Over the years the performance of query engines has greatly improved. Techniques like main-memory optimized data structures, query compilation, just-in-time code ~~gerated~~ generated operators, and more efficient synchronization primitives have led to execution engines that are dramatically faster than previous generation systems. Techniques such as vectorized execution and query-optimized data structures have also been important factors in improving the performance of traditional SQL analytical systems.

There are many hard problems remaining to be solved, especially as both data sizes and query complexities continue to grow. Our current systems are optimized for the simple primitives found in typical SQL queries. Identifying and optimizing for a new set of primitives required by complex analytics remains a challenge in the foreseeable future. Achieving high performance for these new set of primitives is the next important step.

Transaction Processing

Transaction processing has been one of the greatest success stories of the database community. The community defined a set of useful guarantees (e.g., ACID) for arbitrary groups of accesses to different items in the database, and these guarantees have become widely relied upon by large numbers of application developers. The community has led the way in enabling large amounts of concurrency of transactions on single-processor, multi-processor systems, and distributed systems. Modern systems can achieve transactional throughput larger than the vast majority of real-world application requirements, and have continued to refine efficiency and adapt to changing hardware.

One major issue that is present in modern systems is replication over large geographic distances. Such geographically replicated systems present significant challenges for

maintaining consistency and latency given the fundamental limitations in the speed with which data can travel from one part of the world to another. Almost every new system that is developed trades off consistency and latency in a new and different way. There are dozens of different proposed "consistency levels", each one with a different set of guarantees. There is a significant need to unify around a ~~more narrow~~ narrower set of guarantees, as the community did with ACID and the small number of widely implemented isolation levels from a generation earlier.

Another major issue is that the profile of an application developer has evolved over time. Database systems were originally designed for enterprise use cases for businesses that were large and somewhat "slow-moving". The assumption was that prior to application development, the developer had a good sense of how the data would be accessed, and had time to plan in advance, and carefully design a schema for that application. Modern application developers get a "minimum viable prototype" complete as soon as possible, and continuously refine an application after it is already live. There is an increasing need for database systems that evolve with an application.

Data Lakes

Database systems have long contained support for storing and accessing non-structured data, including videos, images, text, and nested data. However, many data scientists prefer to use scalable data processing platforms (such as Hadoop) instead of traditional database systems as a "data lake". The systems provide a scalable file system that can store arbitrary data across a large number of machines in a cluster, and encourage the use of targeted data formats for different data types, that facilitate efficient parallel processing of those data types.

A major accomplishment of the database community has been the integration of parallel database processing techniques for the structured data stored in these data lake platforms. The original versions of Hadoop supported only MapReduce primitives, which lead to large amounts of inefficiency for processing and joining structured data. Projects such as Hive, HadoopDB, Impala, Pig, Presto, Scope, Flink and Spark brought many ideas from the database community to data lakes, including vectorized and column-oriented processing, operating on compressed data, broadcast and co-partitioned joins, declarative queries, and cost-based query optimization.

Unfortunately, data lakes have accentuated the data cleaning, data integration, and data curation problems that have long been studied by the database community. We discuss these problems further in Section 3.

ML for Databases and Autonomic Databases

There are two approaches to using ML to improve DBMSs. The first are methods that alter the runtime behavior of the DBMS's internal components. Examples of such components include the query optimizer's cost model or index data structures. These are written by the developers that built the DBMS and are controlled by rules or heuristics. Thus, instead of relying on these hardcoded rules, new research has explored ways to learn new policies based on the observed workload. The second category is ~~on~~ methods for automatically tuning the configuration of the system, such as the database's physical design, knob configuration, and hardware resources. This is something that DBAs traditionally manage, although there is a long research history of tools to aid them with this process. The goal of this line of work is to relieve humans from many of these laborious tuning tasks, so that they can focus on other, more enriching activities. There is some overlap between these two categories, but the key difference is that the first is about tuning a single component of the system whereas the second is about taking a holistic view of the overall system.

Interoperability across multiple data systems and different type systems (engine aspects)

Interoperability is traditionally a difficult topic. In theory we have a SQL standard that allows for executing queries and exchanging data in a portable manner across systems. In practice, the standard is very weak, leaving many details as "implementation defined". As a consequence, only very simple queries work similarly across systems. Even for the relatively simple TPC-H queries, different systems produce slightly different results due to differences in type inference etc. This situation is largely caused by database vendors that want to fixate their existing behavior and have little interest in improving interoperability.

There has been some momentum from the outside to change that, in particular from the "big data" and Hadoop world. Many systems can nowadays read file formats like RCFile, Orc, or Parquet, which makes data exchange more interoperable. However,

there is some friction between the NF2 nature of some of these file formats and the relational world of most engines.

Unfortunately, data exchange interoperability alone does not help with the query problem. Many existing applications use complex, often hand-crafted abstraction layers that use complex SQL constructs to try to minimize the differences between the systems. This often results in ~~very~~ complex queries that are both hard to optimize and hard to execute.

Impact of hardware and networking trends

Query engines are always ~~adopting~~ adapting to hardware trends. Often, we try to make the best out of hardware designed for others, in particular gaming and ML. Ideally, we should try to guide hardware vendors to better suit our needs. Nonetheless, hardware development is hugely influential for engine design. For example, it is clear that today all systems must scale to a large number of cores. Traditional database systems have already spent much effort, and made much progress on multi-core processing, increasing locality, and minimizing NUMA effects.

Changes in the storage hierarchy also have an important impact on database systems. SSDs are becoming ~~very~~ common as a storage medium, and their fast random I/O and asymmetric write behavior has an impact on systems design. The emerging non-volatile memory has ~~the~~ a chance to disrupt the storage hierarchy even further, as it promises persistent high-density storage with low latency. As this hardware is not widely available yet, and particularly since its price is not known yet, we cannot predict the effect on systems reliably. But there is a general trend towards heterogeneity in hardware, for example by using GPUs or FPGAs as co-processors, and some systems make use of that to offload processing tasks to specialized hardware. The memory hierarchy extends across machines, especially in cloud environments.

For general purpose systems there is some tension between specialization and software complexity. Making use of every available hardware is difficult, in particular if the hardware is not widely used or if the complexity of using it is high. For GPUs there are some existing abstraction layers, but for FPGAs or other specialized hardware, the software is often closely tied to that particular hardware.

Workload management

The trends noted above about data lakes and heterogeneous workloads come with corresponding changes in user behaviour and expectations. Clusters of machines that are viewed as shared resources for storing data and running these workloads are common, and increasingly, users expect workload management features such as fair allocation, managed queues with priorities and access controls, and good resource utilization while guaranteeing SLAs for important, time-critical parts of the workload. Meeting these goals while supporting the heterogeneity in the environment and workloads is challenging.

Open source projects

The database community has ~~had large amounts of~~ success with contributing and maintaining highly sophisticated database systems as open source projects. One great example of this is the PostgreSQL project which originated as a research project in the community and has been used as the core engine for dozens of subsequent commercial database systems and academic research projects. MySQL and its modular design with pluggable storage engines has also been ~~very~~ impactful. Many smaller systems targeted at specific components of a database system, such as BerkeleyDB and LevelDB have also made impact. For analytics, Apache Hadoop has become extremely popular and is used by tens of thousands of end users. One interesting trend is that systems that have started as a proprietary codebase have become open source after achieving maturity (such as the Greenplum analytical database system).

While several good open source single machine transactional and analytical database systems exist, for distributed deployments, only systems targeting ~~for~~ analytical workloads are well covered. There does not yet exist a widely-used distributed, open source, transactional database system that enforces traditional ACID guarantees and strictly serializable isolation. While there does exist open source projects with contributions almost entirely from a single-vendor (such as CockroachDB, TiDB, YugaByte), we have yet to see a truly communal open source distributed transactional database engine.

3. Action items for the database community

Query engine

Research on query engines should continue in multiple dimensions. Scalability and complex analytics remain important challenges for the database community. Today's use cases include ~~very~~ large, complex, machine generated analytical queries, with dozens or even hundreds of relations. These use cases are very challenging both from an execution perspective and also for query optimization. Existing benchmarks do not cover these cases very well, because they concentrate on scaling data size instead of query complexity. It is clear that we could do much better here, both in the engines themselves and also with testing and benchmarking. In the research community, benchmarking should also be considered more critical; experiments are often ~~very~~ artificial, do not report variance or any effects of complex system interactions, and are sometimes hard to reproduce. A more realistic setup, including more realistic data and queries, would be helpful for the community at large. Finally, engines should go beyond pure SQL processing. The analytical processing ~~capability~~ capabilities of database engines are ~~very~~ powerful in many use cases, including "data science" applications, but offering just a SQL interface is not enough for most users. Having a richer interface for applications and users would be essential for bringing these scalable processing capabilities to a much wider usage.

Transaction Processing

Traditional database systems are simply too rigid for modern applications developers. Today's applications evolve rapidly --- the schema changes, data access patterns change, and the scale of access changes. Many application developers have turned to "NoSQL" systems that they believe are more flexible and able to adapt to their rapidly evolving applications. Instead of pointing out the flaws of failing to plan ahead, the database community needs to be better attuned to the needs of the modern application developer and build highly evolvable systems.

"Database systems" is not a required course in most university computer science majors. Furthermore, most application developers do not have a bachelor degree in

computer science. As a result, the vast majority of database system users have no formal training in database systems. Over time, database system users are becoming increasingly less sophisticated. There is thus a need for database systems to become easier to use and more robust to incorrect or suboptimal usage. Hosted database systems in the cloud have reduced complexity for the application developer, but database performance is still very much susceptible to deterioration in the presence of unsophisticated application developers.

It is tempting to state that improving transaction throughput should be a non-goal of the database community, given the extremely high throughputs attainable by modern systems. Although it is true that it is very hard to find an application that requires more throughput than available on a modern, scalable database system, we still predict that in the future, greater throughput will be needed. In today's applications, transactions are usually initiated by humans ~~that~~ and can only transact at the speed a human can act. Tomorrow's applications may have machine-initiated transactions, and need far greater throughput ~~that~~ than what today's transactional systems are able to produce.

Data Lakes

Open source data lakes have made the cost of storing large amounts of data extremely small. As a result, a common use case is to dump data into a data lake without clearly defined curation processes, resulting in a level of chaos akin to a toddler's play room. There is a tremendous need for bringing order to this chaos, and there remains large amounts of "low-hanging fruit" where even imperfect solutions bring significant value.

There is also a need for improved data provenance. Data lakes typically run a data processing algorithm on a large amount of data, and may produce a large amount of data as a result of this operation. This resulting data is stored back in the data lake and may be processed by subsequent operations. Although there do exist good solutions for tracking this provenance at a high level, much work remains to be performed and lower data granularities.

Supporting interoperability across diverse engines operating on heterogeneous data remains challenging, especially with support for effectively sharing the underlying resources while meeting SLAs for key jobs.

Data visualization is also an important problem in this area. Many users have trouble getting a grasp on what data exists in the data lake, what are the properties of different datasets, and how do processing operations affect these properties. As data gets larger, the challenging of getting started with understanding data increases rapidly, and traditional data search techniques are insufficient.

ML for Databases and Autonomic Databases

The early work in this field is mostly about training deep neural networks (DNNs) to replace existing, human-built methods and components. For example, researchers have proposed using DNNs to replace histograms for estimating the distribution of a column's values. Where this work will eventually go is toward how to build systems that use the information collected about databases, workloads, and the system itself to extract patterns that are non-obvious to humans. That means instead of just replacing existing histograms, the system could train a network that can identify weird correlations between tables/columns or incorporate growth trends in its predictions. Future DBMSs can use such "learned" models in conjunction with existing data structures rather than supplanting them entirely. This would allow it to do more than what is possible today with existing techniques. The challenge will be how to maintain these models in a dynamic environment while ensuring performance stability and avoiding regressions if the models go awry.

Interoperability across multiple data systems and different type systems (engine aspects)

Having a stronger SQL- (or in general: query-)standard would be highly desirable. In the programming languages world, nobody would accept that different compilers produce different results for everyday programs. Having a well-defined query semantics that systems agree upon would be incredibly useful for interoperability. Unfortunately, this standard is unlikely to happen. Existing system vendors do not want to change highly complex existing code, and fixing the very large and complex SQL standard is an overwhelming endeavor. The best chance to improve interoperability is to think beyond SQL. Users increasingly want to express their algorithms in some form of programming language, and they will need some well defined semantics for that. Getting the

semantics standardized now with a robust standard will allow engines to expose interfaces with better interoperability.

The challenge is to find a good compromise between expressiveness, ease of use, optimizability, and efficiency of execution. For example, some users may want to express their algorithms in Python, which has well defined semantics and is widely known, but is usually not declarative and not efficient to execute. The Spark DataFrames are more database-like, but still not nearly as declarative as SQL. Finding a good compromise is challenging, but if successful will be very impactful.

Furthermore, it is instructive to think about the applications themselves as another "data system", with its own type system, access pattern, impedance mismatch, and so on. For users it would be highly desirable to blur the distinction between query engine and application, allowing easy and seamless access to the underlying database system.

Impact of hardware trends

Adapting to changing hardware is a must, as systems design is often driven by what the hardware offers as functionality. One particular interesting ongoing trend is the development in SSD storage. SSDs have been around for quite a while, but in recent years both the price per GB and access latencies have improved dramatically. In fact, performance has improved so much that it is really a game changer compared to older SSDs. With modern high-speed SSDs, it becomes feasible to challenge a pure in-memory system, with a much better performance per dollar. These changes are very important because systems design for a low-latency SSD look quite different from a system aiming at rotating disks.

An interesting alternative is the upcoming non-volatile memory that promises even better latency, albeit presumably at a higher price point. Systems aiming for NVM usage would again look quite different from systems aiming at SSDs. In the long run it might be beneficial to combine both, for example using fast SSDs for storage and NVM for logging and recovery, as that promises very good performance at a reasonable price. Similarly, thinking of remote storage sub-systems with fast networking offers promise.

]

The increasing heterogeneity in hardware is a significant challenge to systems design. Re-implementing large parts of the system whenever new hardware comes up is not a sustainable option, and different users will have different combinations of specialized hardware available. It is necessary to develop abstraction layers that allow for expressing high-level algorithms across these different devices in unified way, and that lead to efficient execution on whatever hardware is available. This is a hard problem because the different hardware devices behave ~~very~~ differently, and, e.g., a CPU and a GPU will favor different execution modes. But nevertheless such abstractions are necessary to cope with the upcoming zoo of specialized hardware.

Section 4 discusses the relationship with the architecture community for affecting hardware development trends.

Open source projects

As mentioned above, there is a significant gap in community developed, open source, distributed ACID transactional database systems that support strictly serializable isolation. It is important that the community unify around an open source project in this space in the upcoming years.

4. Relationship to other Research Communities

Query engine

There are, or should be, close ties between query engines and the systems community. These groups are often looking at very similar problems, in particular concerning the interactions between the query engine, the hardware, and the operating system. In practice these ties could be stronger. There are a few people that are active in both communities, but still the communities are often separated. Also from a practical perspective it would be highly desirable to have better interaction between engines and operating systems. For example, it is very hard and inefficient to guarantee durability with today's file system abstractions, even though the underlying hardware could often provide that efficiently. But that functionality is not exposed by any mainstream OS. Another neighboring field is the programming languages community. When query engines go beyond SQL, programming languages become very important. Also, the programming languages community itself can benefit from a better interaction with

databases: Mainstream programming languages like C++ currently work on exposing data parallelism in a natural way within the programming language itself. This is highly related to how database query languages work, and, as additional benefit, makes it easier to execute the program fragments within the query engine itself.

Transaction Processing

The OSDI, SOSP, and NSDI communities have published more papers on geographically replicated database systems than the database community. Several highly influential papers published by those communities are not cited frequently enough by papers in SIGMOD or VLDB. The database community needs to do a better job of building on top of the pioneering work in this area from these other communities.

Data Lakes

Many of the data lake challenges mentioned above concern interaction of humans with data. Collaboration and cross-pollination with the HCI community is important for the development of usable tools in this area.

Interoperability across multiple data systems and different type systems (engine aspects)

The programming languages community is working on very similar problems, in particular for data parallel processing. There are people that are active in both communities, for example the DBPL workshop has been ongoing for a long time now. However, there is a danger that the programming language community will pick something from their side without any input from the database community. A stronger interaction between these two groups is highly desirable, and would benefit both sides.

Impact of hardware trends

Hardware/software co-design would be ideal to cope with these hardware trends. Query engines will adapt to whatever hardware is available, but if specialized hardware is coming, it can as well try to support frequent data processing operations more efficiently. Examples for that could be efficient vectorized gather support or asynchronous bulk hash table lookups. These operations occur very frequently during query processing, and make a significant part of execution costs, and thus would be particularly attractive to have as specialized circuits.

On the other hand data processing engines are not a mass market compared to consumer devices, thus it is not realistic to expect specialized circuits in upcoming chips. Unless these specialized operations would be useful in more generic programs, too, which could be the case for common operations like hash table lookups. In addition there are many constraints imposed by the underlying architectures. For example increasing the number of load ports in a CPU would be highly attractive for many data processing tasks, but doing so is nearly impossible in current mainstream architectures.

The hardware community and the database community should discuss what is desirable from the software side and what is feasible from the hardware side, trying to find useful and commonly used functionality.

Similar for storage devices, where interaction between hardware vendors and data processing engines could be very fruitful. Open-channel SSDs are an example where high-level logic can be brought closer to the hardware, where some database tasks can be implemented much more efficiently.

Recommendations

- Concentrate on weaknesses of current engines
- Use cases beyond SQL
- Hardware trends have a large impact on design
- Abstractions for hardware zoo required
- Win mind share with both performance and ease of use
- Should consider the **federated case** in which different organizations, cloud or user devices retain control over data (or are confined by regional laws and privacy regulations) and want to do federated query processing

Discussion Points

- Are engines solved? Easy problems vs. hard challenges
- Use cases in and beyond SQL

- Usability and silos remain a huge problem
- Interaction with other communities
- Hardware trends, input from DB community
- One size doesn’t fit all vs. silos and ease of use

Scope: Sub-topics (not exhaustive and based on survey) that should be addressed

- Query engine
- Transaction Processing
- Data lakes
- Interoperability across multiple data systems and different type systems (engine aspects)
- ML for Database Internals
- Impact of hardware trends
- Open-source

1. **Introduction**

2. **Our Accomplishments/Score Card so far**
   What is the database community already doing to address these problems?

3. **Call to Action: What should the database community be doing to address these problems?**
   What should be the goal of the database community? What are non-goals?

4. **Relationship to other Research Communities**
   Discuss how the database community should manage relationship with other communities that work in this area

Appendix

Survey notes

Transaction Processing Applications

- Transaction processing (both multicore and distributed)
- Transaction processing
- Rethinking the boundaries between the application and the database (ORMs, SQL, txns).
- Mapping, understanding and navigating the design space of data structures to accelerate research and engineering productivity and to enable a new class of systems with deep adaptivity. Special focus on NoSQL k-v stores which is an untapped opportunity for our community.
- Data base design (customers have trouble with that). Interaction between data base design and application design

- Schema evolution
- Improving the ability of programmers to build data-centric systems and solutions: data-centric programming, software synthesis and beyond. → [Note: This topic can be here or under data analytics]

Data Lakes and Heterogeneous Data Sources and Types

- Data lakes and generally the management of heterogeneous types of data (videos, images, text, structured, etc.).
- Video Data Analytics
- Of course, a closely related trend is that users want to store and query very diverse and distributed datasets. Hadoop-style ""RISC"" architectures for plugging in multiple engines via standard resource management and storage APIs such as HDFS and YARN is a good first step, but what does the future hold? Deeper integration with single engines capable of effective performance across files and pre-loaded tables?
- Analytics of semi-structured data.
- Data analytics across data centers
- Lack of functionality (support for less conventional data types and operators)
- Graph DB

ML for Databases and Autonomic Databases

- Auto-tuning/self-managed databases
- Autonomous Systems
- Using ML to improve Data Management Systems (ML for Systems). DB+ML. Machine learning and data management. ML is currently overhyped. integrating ML into relational engines. Machine learning. Opportunities for leveraging ML for Data Systems Stack. Leveraging machine learning techniques to solve data problems. Database machine learning. Learned index structures. DB's and machine learning (applying ML to DB problems).
- "Automated tuning of the traditional data stacks using machine learning.
- Instance-optimized Database Systems (AI for Database Systems).
- Using solvers for reasoning about queries and constraints (2) incorporating causal reasoning in declarative queries
- Fusing AI and DB

Hardware trends

- GPGPUs, NVM
- Hardware acceleration and the changing hardware landscape
- Methods to allow data platform to exploit the full potential of the dramatically different hardware that is anticipated in the near future.

- Non-Volatile Memory
- Data management on heterogeneous hardware,
- How can we best leverage rapid evolution in hardware (e.g., FPGAs, GPUs, RDMA, NVME) and software (e.g., NNs) as we expand the surface area of analytics that is natively supported in the DBMS? Can we blend advances in multi-core processing with the massively distributed scale-out architectures for big data?
- New storage media and impacts on DBMS architecture.
- Impact of modern hardware on Data Systems Stack
- Shared-Disk Architectures
- Impact of new hardware on database systems
- Co-design of DB software and underlying hardware (especially for the cloud).
- Easy tailoring to specific context/data/hardware. Understanding how a system behaves or would behave if the conditions/hardware/workload change.

Usability

- Ease-of-use, Reducing Complexity. Easy of use.
- Is the problem of complexity of choosing and configuring data management systems only getting harder with the increased number of choices of tools we have today? I don't know this for a fact, but it seems plausible and worth discussing.
- Usability for end-users, where the end-users are not DBA or programmers but "common" people.
- Scalability and usability
- Providing guidance to them. They are facing a vast number of tools and platforms, and when faced with a problem, they often do not even know where to begin. We are very good at developing technical solutions for point problems. But we really suck at providing guidance to users on how to solve the whole problem, end to end.
- we need easier to use data platforms that don't require a PhD to install and ingest data!
- fragmented data, too many tools,
- Dealing with change management across the stack

Open Source Projects

- The discussion I think is worth having is how to encourage the community to develop and rally around open-source projects so they have significant impact whether it's in data integration or other areas of data management. There is not enough work on system building and reaching out to communities that actually build practical systems, such as the PyData/R community.
- Lack of a good open-source distributed OLTP DBMS. This is the one thing that I have heard from companies multiple times in the last three years.

Working Group Participants:

Please list your name here, and add some notes about which content your are interested in.

Beng Chin Ooi -- it is good to use h/w to accelerate, but it is making the system more and more specific to certain h/w?

Sam madden - transactions / analytics, video, AI for databases. As a general topic I'd like to see some discussion as to whether the academic community is focused on the right metrics. Do we really need more TPS?

AnHai Doan: I'm interested in data lakes, interoperability across data systems, ML for db internals, and usability and open source.

C. Mohan: HTAP, Blockchain + Databases, Recovery, ...

Andy Pavlo: Autonomic/Self-Driving Databases

Dave Andersen: ML for Databases, hardware trends,

Jignesh Patel:

- Simply measuring performance needs to be rethought.
    - We can't just do raw latency/throughput as that is not a good measure in practice. But, our paper use these simple methods. Heck our papers don't even report variance. Seems like for a community that cares about performance a lot, we really don't know how to measure performance. Boxplots at the least everywhere! (BTW, some reviewers get very nervous when they see a box plot, so may need to educate folks on 101 in stats. Reviewers of performance papers must take a quiz perhaps before being declared experts? :-)
    - We can do some $/DBUNIT (e.g. Sort benchmark's Cloud category) but that is meaningless as any economist will tell you -- you need to worry about scale. If I get a low cost but have to have a PB DB and 1000 nodes install, that is not what most DB deployments look like. They are far smaller.
    - TPC does a nicer job of breaking down by scale-factors, but misses the point that you may a limit on the budget so you can't spend $100K to start with.
    - End-to-end cost also includes other costs associated with downtime. All this means we need to start looking at how to measure performance in a new way.
    - Single metric may not be enough, but we need to move away from just raw latency.
    - Reporting at Nth percentile (N=95 or 99) is really important in cloud settings
    - Need a good benchmark with modern metrics that the community can fall behind. This benchmark should allow "regular" folks to play and not need to provision a 100 node machine by default.
- We as a community are poor in telling what relational DBMSs can do, and get side-swiped from outside by crazy ideas (that are going to production!) on how to build systems for specialized tasks (e.g. Graph databases, document stores …) We should

also work on making relational system not tied to SQL. That is killing us. The core relational engine can do a lot more, it's not just a SQL engine. Widen the app surface!

- We need to integrate text search far better inside a database engines that do other structured query processing (using RA) well. Lots of ideas but the world still does text search outside a DBMS. No reason it should.
- We have overemphasised in the last decade "scaling out." This is complementary to scaling-in, which involves making the most of the single nodes that are massive SMPs today and in the near future will be like a small datacenter with a mix of SN and SMP topologies prebuilt. Got to go back to the core. Do we really know know to use ~100 core machines available today? 1000 cores?
    - Think both single query and multiple concurrent queries. For OLAP the single-query case in important in real-time analytics applications.
- NVM is poised for disruption. There will invariably be a cottage industry that incorrectly call for new solutions when new solutions are not needed. Can we be smarter about this? See benchmarks above for one aspect, but the other is a good reference implementation using "good-old-techniques."

Stratos Idreos: Creating the Genome of Data Systems so that we can finally have some kind of understanding on how to design systems and why they behave the way they do, when they would break with a new feature, how to adopt new hardware and so on. Right now we do all system design manually which does not scale. Once we have a Genome, we can employ machine learning, solvers or new search techniques to build systems that self-design, i.e., the build the optimal or close to optimal storage and algorithms for the specific workload and hardware. Specific applications that I think are interesting in the next few years are NoSQL key-value stores, HTAP and storage for ML algorithms. Hardware software co-design can also bring very interesting new solutions that go beyond what we can do only with smart algorithms. Performance does not have to be the only metric: e.g., energy and memory amplification are crucial and can drive system design.

Alvin Cheung: compilation and execution techniques for queries. Should queries be interpreted or compiled? What are the implications given new hardware trends? Also, DBs for non textual data, given that most of today's data is in the form of images and videos.

Raghu Ramakrishnan: Cut-and-paste from my email:

- How does the recent work in multi-core DB engine architectures (including this years SIGMOD PhD award) compose with the scale-out architectures for big data?

- What is the implication of the evolution of warehouses to more heterogeneous data collections with a richer mix of analytics (including streaming/batch/interactive along one dimension, and ML/log analytics and text processing alongside TPC-H kind of workloads along another dimension)? Increasingly users seem to prefer engines that cover a larger surface area as opposed to mixing and matching engines, and covering this broad spectrum presents novel challenges. And the datasets are getting order of magnitude larger, thanks to scenarios like telemetry and IoT ...

- What is the implication of the movement to cloud? This means HW is much less reliable than custom on-prem clusters; it means lots of commodity VMs are the cost-effective way to go; elasticity poses interesting challenges wrt caching strategies; ...

Looks like you guys already touched on most of this. I think a lot of the comments about changes in the HW landscape are spot on, and would just add my comment about cloud to this broader observation that the execution environment is being disrupted significantly.

Gustavo Alonso:

- We need to develop a more realistic understanding of performance and functionality. There should be a written record of what can be done so that there is publicly available information on what different systems can do under what constraints. It would also help to raise the profile of the conferences if that would be on of their regular outcomes (information relevant to database users and to companies).
- Hardware is changing everything: processors, memory, interconnects, networking, etc. It is surprising how the academic community largely ignores the fact that modern hardware plays a huge role in the performance of a database engine.
- Database engines can or could do more than SQL. This is an area that needs significant attention (topic related to the data science working group)

Joe Hellerstein:

- On trick I learned and now adhere to is to define and compare to "speed-of-light" performance goals for subsystems based on HW capabilities -- what is the fastest we could *possibly* go on this hardware. Then see how close we can get. Papers could report how close they are to the HW potential. For core tasks (e.g. record storage/retrieval, scans/sorts) you can model this pretty accurately. Then of course the challenge is to figure out whether you lose that win as you look at an end-to-end system. We did this internally in the Anna KVS work and it helped us get to an order or 2 of magnitude better than earlier systems for some workloads, but also exposed to us that once we nailed that, the bottleneck moved to the bandwidth of ethernet cards for handling remote client traffic. (BTW, sometimes you realize your "speed of light" model embedded assumptions that can be bypassed and you can go even faster ... that's a good discovery in itself, and simply causes more iteration on this process.)
- Autoscaling and cost are critical metrics today in environments with shared resources -- primarily the cloud but also large private datacenters with virtualization ("private clouds"). Autoscaling opens up multidimensional tradeoffs between cost metrics and performance metrics. SLOs may be the right way to measure systems going forward. We need accepted metrics/benchmarks around this.

Dan Suciu

- A neglected topic seems to be heavy-duty database support for ML. By heavy-duty I mean pushing the entire ML task down the transformation/integration pipeline. Typical

entreprise ML applications start by constructing the “design matrix”, which usually means joining together all the relations that have any data of interest, then running some standard ML algorithm on the universal relation. I know that in retail this is the common scenario, but I’m sure others domains also work this way. The DB community has the know-how to push down the ML algorithms to the source tables: but this is not easy, since the ML algorithm often has lots of aggregates (x100, x1000), and may involve other operations that we don’t understand yet how to push down.

Anastasia Ailamaki:

- Interested in just-in-time data management, codegen, data virtualization. AI seeps through to the core of how we build these systems, in a genuinely constructive way that changes the paradigm toward lean and agile Stacks.
- I support Jignesh’s point about the quality of measurement methodologies 100%. We should educate our community. Part of why the systems community suffers so much is that our credibility is lowered by sloppily done experiment. A closer look at and cross-reference of numbers often reveals that they don’t hold water!
- Joe’s first point is about “envelope”—a technique which should be applied to all systems studies. What is the theoretical max we can push things to on a certain microarchitecture? Then evaluate the state of the art based on how close it is to that, and see if it is worth taking things further – before we change the microarchitecture. That’s hard to do but the learning benefit is huge!
- We have A LOT to learn from the systems community (and vice versa). It almost seems that we have a wealth of big problems, and they come up with better solutions! A good idea may be to co-host top DB and systems conferences every couple of years, to help exposure, communication and exchange of ideas.
- I’ve always worked on hardware conscious data management, but this nowadays needs to be done through hardware oblivious ways which employ late binding of alternatives to HW resources at hand. This way we can ensure utilization on cloud.