Large Synoptic Survey Telescope (LSST)
Database Design

Jacek Becla, K-T Lim, Daniel Wang

LDM-135

08/12/2011
## Change Record

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Description</th>
<th>Revision Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>6/15/2009</td>
<td>Initial version</td>
<td>Jacek Becla</td>
</tr>
<tr>
<td>2.0</td>
<td>7/12/2011</td>
<td>Most sections rewritten, added scalability test section</td>
<td>Jacek Becla</td>
</tr>
<tr>
<td>2.1</td>
<td>8/12/2011</td>
<td>Refreshed future-plans and schedule of testing sections, added section about fault tolerance</td>
<td>Jacek Becla, Daniel Wang</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Executive Summary.................................................................................................................... 7
2. Introduction................................................................................................................................. 8
3. Baseline Architecture.................................................................................................................. 9
   3.1 Alert Production and Up-to-date Catalog............................................................................ 9
   3.2 Data Release Production......................................................................................................... 9
   3.3 User Query Access.................................................................................................................. 10
      3.3.1 Distributed and Parallel............................................................................................... 10
      3.3.2 Shared-nothing.............................................................................................................. 10
      3.3.3 Indexing........................................................................................................................ 11
      3.3.4 Shared Scanning............................................................................................................ 11
      3.3.5 Clustering...................................................................................................................... 12
      3.3.6 Partitioning.................................................................................................................. 12
      3.3.7 Technology Choice....................................................................................................... 14
4. Requirements.................................................................................................................................. 15
   4.1 General Requirements............................................................................................................ 15
   4.2 Data Production Related Requirements................................................................................. 16
   4.3 Query Access Related Requirements.................................................................................... 17
   4.4 Discussion.............................................................................................................................. 18
5. Potential Solutions - Research..................................................................................................... 19
   5.1 The Research......................................................................................................................... 19
   5.2 The Results........................................................................................................................... 20
   5.3 Map/Reduce-based and NoSQL Solutions............................................................................. 21
   5.4 DBMS Solutions.................................................................................................................... 22
      5.4.1 Parallel DBMSes............................................................................................................. 22
      5.4.2 Object-oriented solutions............................................................................................... 23
      5.4.3 Row-based vs columnar stores...................................................................................... 24
      5.4.4 Appliances...................................................................................................................... 26
   5.5 Comparison and Discussion.................................................................................................... 26
6. Design Trade-offs....................................................................................................................... 31
   6.1 Standalone Tests..................................................................................................................... 31
      6.1.1 Spatial join performance............................................................................................... 31
      6.1.2 Building sub-partitions............................................................................................... 31
      6.1.3 Sub-partition overhead................................................................................................. 32
6.1.4 Avoiding materializing sub-partitions........................................................................32
6.1.5 Billion row table / reference catalog.......................................................................32
6.1.6 Compression................................................................................................................32
6.1.7 Full table scan performance..........................................................................................32
6.1.8 Multi-node partitioning overheads.................................................................................33
6.1.9 Low-volume queries.......................................................................................................33
6.1.10 Solid state disks............................................................................................................34
6.2 Data Challenge Related Tests..........................................................................................34
   6.2.1 DC1: data ingest..............................................................................................................35
   6.2.2 DC2: source/object association......................................................................................35
   6.2.3 DC3: catalog construction..............................................................................................35
   6.2.4 DC4: end user query/L3 data production......................................................................35
7. Risk Analysis.........................................................................................................................35
   7.1 Potential Key Risks..........................................................................................................35
   7.2 Risks Mitigations.............................................................................................................37
8. Implementation of the Query Access Prototype (Qserv)......................................................38
   8.1 Components....................................................................................................................38
      8.1.1 MySQL.......................................................................................................................38
      8.1.2 Xrootd.........................................................................................................................38
   8.2 Partitioning......................................................................................................................39
   8.3 Query Generation.............................................................................................................40
   8.4 Dispatch..........................................................................................................................41
   8.5 Aggregation......................................................................................................................42
   8.6 Indexing...........................................................................................................................42
   8.7 Cluster and Task Management.........................................................................................42
   8.8 Fault Tolerance................................................................................................................42
   8.9 Current Status and Future Plans......................................................................................43
9. Large-scale Testing................................................................................................................48
   9.1 Introduction......................................................................................................................48
      9.1.1 Ideal environment.........................................................................................................48
      9.1.2 Schedule of testing......................................................................................................49
      9.1.3 Current status of tests.................................................................................................49
   9.2 150-node Scalability Test................................................................................................50
      9.2.1 Hardware....................................................................................................................50
      9.2.2 Data............................................................................................................................50
      9.2.3 Queries.......................................................................................................................50
9.2.3.1 Low-volume 1 – object retrieval ..................................................51
9.2.3.2 Low-volume 2 – time series .......................................................51
9.2.3.3 Low-volume 3 – spatially-restricted filter ....................................52
9.2.3.4 High volume 1 – count ...............................................................52
9.2.3.5 High-volume 2 – full-sky filter ....................................................53
9.2.3.6 High-volume 3 – density ..............................................................54
9.2.3.7 Super-high-volume 1 – near neighbor ........................................54
9.2.3.8 Super-high-volume 2 – sources not near objects .........................55
9.2.4 Scaling .........................................................................................55
9.2.5 Concurrency .................................................................................57
9.2.6 Discussion ...................................................................................58
10. References .......................................................................................60
11. Appendix A – Map/Reduce Solutions ..................................................64
   11.1 Hadoop .......................................................................................64
   11.2 Hive ............................................................................................66
   11.3 HBase ........................................................................................66
   11.4 Pig Latin .....................................................................................66
   11.5 Other Hadoop-related Systems ....................................................66
   11.6 Dryad ..........................................................................................67
   11.7 Dremel ........................................................................................68
   11.8 Tenzing .......................................................................................68
   11.9 "NoSQL" ......................................................................................68
12. Appendix B – Database Solutions .......................................................68
   12.1 Caché .........................................................................................69
   12.2 DB2 .............................................................................................69
   12.3 Drizzle .........................................................................................69
   12.4 Greenplum ..................................................................................70
   12.5 GridSQL .......................................................................................70
   12.6 InfiniDB ......................................................................................72
   12.7 LucidDB ......................................................................................72
   12.8 MySQL .......................................................................................72
   12.8.1 MySQL – Columnar Engines ......................................................73
   12.9 Netezza .......................................................................................74
   12.10 Oracle .......................................................................................74
   12.11 ParAccel .....................................................................................74
   12.12 PostgreSQL ...............................................................................75
   12.13 SciDB .......................................................................................76
12.14 SQLServer......................................................................................................................76
12.15 Sybase IQ.....................................................................................................................77
12.16 Teradata..........................................................................................................................77
12.17 Vertica............................................................................................................................77
12.18 Others..............................................................................................................................78
   12.18.1 Cluster and task and management...........................................................................78
13. Appendix C: Tests with InfiniDB........................................................................................78
14. Appendix D Qserv-related Research Topics.....................................................................92
15. Appendix E: People/Communities We Talked To..............................................................92
1. Executive Summary

The LSST baseline database architecture is an MPP (massively parallel processing) relational database composed of a single-node non-parallel DBMS, a distributed communications layer, and a master controller, all running on a shared-nothing cluster of commodity servers with locally attached spinning disk drives, capable of incremental scaling and recovering from hardware failures without disrupting running queries. All large catalogs are spatially partitioned into materialized *chunks*, and the remaining catalogs are replicated on each server; the chunks are distributed across all nodes. The Object catalog is further partitioned into *sub-chunks* with overlaps, materialized on-the-fly when needed. Chunking is handled automatically without exposure to users. The system relies heavily on indexes to speed up spatial searches, time series analysis, and simple but interactive queries. Shared scans are used to answer all but the interactive queries. Such an architecture is primarily driven by the variety and complexity of anticipated queries, ranging from single object lookups to complex \( O(n^2) \) full-sky correlations over billions of elements.

Given the current state of the RDBMS and Map/Reduce market, an RDBMS-based solution is a much better fit, primarily due to features such as indexes, schema and speed; however the entire Map/Reduce community (both Google’s proprietary system and open source Hadoop) has recently turned their attention to adding RDBMS features and will likely soon catch up. No off-the-shelf, reasonably priced solution meets our requirements (today), even though production systems at a scale comparable to LSST have been demonstrated already by industrial users such as eBay using a prohibitively expensive, commercial RDBMS.

The baseline design involves many choices such as component technology, partition size, index usage, normalization level, compression trade-offs, applicability of technologies such as solid state disks, ingest techniques and others. We ran many tests to determine the design configuration, determine limits and uncover potential bottlenecks. In particular, we chose MySQL as our baseline open source, single-node DBMS and Xrootd as an open source, elastic, distributed, fault-tolerant messaging system.
We developed a prototype to implement the baseline architecture, called *Qserv*. To mitigate major risks, such as insufficient scalability or potential future problems with the underlying RDBMS, *Qserv* pays close attention to minimizing exposure to vendor-specific features and add-ons. Many key features including the scalable dispatch system and 2-level partitioner have been implemented at the prototype level and integrated with the two underlying production-quality components: MySQL and Xrootd. Scalability and performance have been successfully demonstrated on a 150-node cluster using a 30TB data set and tables as large as 50 billion rows, or ~10% of the first LSST data release. Required data rates for all types of queries (interactive, full sky scans, joins, correlations) have been achieved. Limited concurrency at scale was demonstrated. Future work involves implementing shared scans, connecting retry logic with the Xrootd fault tolerance recovery mechanisms, demonstrating cross-matching, various non-critical optimizations, and most importantly, making the prototype more user-friendly and turning it into a production-ready system.

If an equivalent open-source, community supported, off-the-shelf database system were to become available, it could present significant support cost advantages over a production-ready *Qserv*. To increase the chances such a system will become reality in the next few years, we initiated the SciDB array-based scientific database project and work closely with its development team. In addition we closely collaborate with the MonetDB open source columnar database team – building on our *Qserv* lessons-learned, the MonetDB team is trying to add missing features and turn their software into a system capable of supporting LSST needs. A demonstration is expected in late 2011. Further, to stay current with the state-of-the-art in peta-scale data management and analysis, we continue a dialog with all relevant solution providers, both DBMS and Map/Reduce, as well as with data-intensive users, both industrial and scientific, through the XLDB conference and workshop series we lead, and beyond.

2. **Introduction**

This document discusses the LSST database system architecture. Chapter 3 discusses the baseline architecture. Chapter 4 explains the LSST database-related requirements. Chapter 5 covers in-depth analysis of existing off-the-shelf potential solutions (Map/Reduce and RDBMS). Chapter 6 discusses design trade-offs and decision process, including small scale tests we run. Chapter 7 covers risk analysis. Chapter 8 discusses the prototype design (*Qserv*), including design, current status of the software and future plans. Chapter 9 describes large scale *Qserv* tests.
3. Baseline Architecture

This chapter describes the most important aspects of the LSST baseline database architecture. The choice of the architecture are driven by the project requirements (see chapter 4.) as well as cost, availability and maturity of the off-the-shelf solutions currently available on the market (see chapter 5.), and design trade-offs (see chapter 6.). The architecture is periodically revisited: we continuously monitor all relevant technologies, and accordingly fine-tune the baseline architecture.

In summary, the LSST baseline database architecture is an MPP (multi-processor, parallel) relational database running on a shared-nothing cluster of commodity servers with locally attached spinning disk drives; capable of (a) incremental scaling and (b) recovering from hardware failures without disrupting running queries. All large catalogs are spatially partitioned into materialized *chunks*, and the remaining catalogs are replicated on each server; the chunks are distributed across all nodes. The Object catalog is further partitioned into *sub-chunks* with overlaps, materialized on-the-fly when needed. Shared scans are used to answer all but low-volume user queries. Details follow below.

3.1 Alert Production and Up-to-date Catalog

The catalog for Alert Production include two large catalogs: Object, DiaSource; the two largest catalogs, ForcedSource and Source are not needed.

We expect to maintain two independent sets of catalogs: one for production and one for querying, and swap the roles of each catalog every night. This approach isolates production activities from user queries. Only short-running queries will be allowed—long running queries would prevent making the system quiescent prior to swapping the roles. Avoiding long-running queries is possible because we don't need to maintain the largest catalogs (Source, ForcedSource) for the Alert Production or Up-to-date Catalog.

Given the relatively small rate of updates, the updates, including updates to the indexes will be done on-the-fly.

3.2 Data Release Production

During data release production, the pipelines will have no direct access to the database. Data ingest from pipelines happens through ASCII CSV files. These fill are ingested into database tables in parallel, directly into pre-partitioned tables, in a dedicated, post-processing phase. This stage will also involve building indexes, and optimizing structured for user query access.
Querying database from pipeline is simple and non-challenging, in particular when compared to user query access: we expect several full table scans over the course of the entire Data Release Production (several months), while user query access will need to deliver many such scans on daily/weekly bases.

3.3 User Query Access

The user query access is the primary driver of the scalable database architecture. Such architecture is described below.

3.3.1 Distributed and Parallel

The database architecture for user query access relies on a model of distributing computation among autonomous worker nodes. Autonomous workers have no direct knowledge of each other and can complete their assigned work without data or management from their peers. This implies that data must be partitioned, and the system must be capable of dividing a single user query into sub-queries, and executing these sub-queries in parallel – running a high-volume query without parallelizing it would take unacceptably long time, even if run on very fast CPU. The parallelism and data distribution should be handled automatically by the system and hidden from users.

3.3.2 Shared-nothing

LSST's catalog will involve petabytes spread over many nodes. Operating under the assumption that locally-attached storage always has the highest bandwidth per unit cost, a shared-nothing architecture allows each node to focus on its own work on its own data. There is no contention at a shared storage apparatus for bandwidth or IOPS, and there is no wasted time spent coordinating other than receiving work and returning results. Interconnection fabric can be constructed of simple low-cost commodity networking hardware without specialized high-bandwidth or low-latency hardware.
Such architecture provides good foundation for incremental scaling and fault recovery: because nodes have no direct knowledge of each other and can complete their assigned work without data or management from their peers, it is possible to add node to, or remove node from such system with no (or with minimal) disruption. However, to achieve fault tolerance and provide recover mechanisms, appropriate smarts have to be build into the node management software.

3.3.3 Indexing

Disk I/O bandwidth is expected to be the greatest bottleneck. Data can be accessed either through index, which typically translates to a random access, or a scan, which translates to a sequential read (unless multiple competing scans are involved).

Indexes dramatically speed up locating individual rows, and avoid expensive full table scans. They are essential to answer low volume queries quickly, and to do efficient table joins. Also, spatial indexes are essential. However, unlike in traditional, small-scale systems, the advantage of indexes become questionable when a larger number of rows is to be selected from a table. In case of LSST, selecting even a 0.01% of a table might lead to selecting millions of rows. Since each fetch through an index might turn into a disk seek, it is often cheaper to read sequentially from disk than to seek for particular rows via index, especially when the index itself is out-of-memory. For that reason the architecture forgoes relying on heavy indexing, only a small number of carefully selected indexes essential for answering low-volume queries, enabling table joins, and speeding up spatial searches will be maintained.

3.3.4 Shared Scanning

Now with table-scanning being the norm rather than the exception and each scan taking a significant amount of time, multiple full-scan queries would randomize disk access if they each employed their own full-scanning read from disk. Shared scanning (also called convoy scheduling) shares the I/O from each scan with multiple queries. The table is read in pieces, and all concerning queries operate on that piece while it is in memory. In this way, results from many full-scan queries can be returned in little more than the time for a single full-scan query.

Shared scanning will be used for all high-volume and super-high volume queries. Shared scanning is helpful for unpredictable, ad-hoc analysis, where it prevents the extra load from increasing the disk /IO cost – only more CPU is needed. On average we expect to continuously run the following scans:

• at least one full table scans of an Object table (the most frequently accessed table),
• one synchronized full table scan of Object, Source and DiaSource tables every 16 hours,
• one synchronized full table scan of Object and ForcedSource table every 7 days.
Shared scans will take advantage of table chunking explained below. In practice, within a single node a scan will involve fetching sequentially a chunk of data at a time and executing on this chunk all queries in the queue. The level of parallelism will depend on the number of available cores.

Running multiple shared scans allows relatively fast response time for Object-only queries, and supporting complex, multi-table joins: synchronized scans are required for two-way joins between different tables. For a self-joins, a single shared scans will be sufficient, however each node must have sufficient memory to hold 2 chunks at any given time (the processed chunk and next chunk). Refer to the sizing model [1.] for further details on the cost of shared scans.

Low-volume queries will be executed ad-hoc, interleaved with the shared scans. Given the number of spinning disks is much larger than the number of low-volume queries running at any given time, this will have very limited impact on the sequential disk I/O of the scans, as shown in [1.].

3.3.5 Clustering
The data in the Object Catalog will be physically clustered on disk spatially – that means that objects collocated in space will be also collocated on disk. All Source-type catalogs (Source, ForcedSource, DiaSource, ForcedDiaSource, CalibSource) will be clustered based on their corresponding objectId – this approach enforces spatial clustering and collocates sources belonging to the same object, allowing sequential read for queries that involve times series analysis.

3.3.6 Partitioning
Data must be partitioned among nodes in a shared-nothing architecture. While some sharding approaches partition data based on a hash of the primary key, this approach is unusable for LSST data since it eliminates optimizations based on celestial objects' spatial nature.

Sharded data and sharded queries
All catalogs that require spatial partitioning (Object, Source, ForcedSource, DiaSource, ForcedDiaSource, CalibSource) as well as all the auxiliary tables associated with them, such as ObjectType, or PhotoZ, will be divided into spatial partitions of roughly the same area by partitioning then into declination zones, and chunking each zone into ra stripes. Further, to be able to perform table joins without expensive inter-node data transfers, partitioning boundaries for each partitioned table must be aligned, and chunks of different tables corresponding to the same area of sky must be co-located on the same node. To ensure chunks are appropriately sized, the two largest catalogs, Source and ForcedSource, are expected to be partitioned into finer-grain chunks. Since objects occur at an approximately-constant density throughout the celestial sphere, an equal-area partition should spread a load that is uniformly distributed over the sky.

Smaller catalogs that can be partitioned spatially, such as Alert and exposure metadata will be partitioned spatially. All remaining catalogs, such provenance or SDQA tables will be replicated on each node. The size of these catalogs is expected to be only a few terabytes.

With data in separate physical partitions, user queries are themselves fragmented into separate physical queries to be executed on partitions. Each physical query's result can be combined into a single final result.

**Two-level partitions**

Determining the size and number of data partitions may not be obvious. Queries are fragmented according to partitions so an increasing number of partitions increases the number of physical queries to be dispatched, managed, and aggregated. Thus a greater number of partitions increases the potential for parallelism but also increases the overhead. For a data-intensive and bandwidth-limited query, a parallelization width close to the number of disk spindles should minimize seeks and maximizing bandwidth and performance.

From a management perspective, more partitions facilitate rebalancing data among nodes when nodes are added or removed. If the number of partitions were equal to the number of nodes, then the addition of a new node would require the data to be re-partitioned. On the other hand, if there were many more partitions than nodes, then a set of partitions could be assigned to the new node without re-computing partition boundaries.

Smaller and more numerous partitions benefit spatial joins. In an astronomical context, we are interested in objects near other objects, and thus a full $O(n^2)$ join is not required—a localized spatial join is more appropriate. With spatial data split into smaller partitions, an SQL engine computing the join need not even consider (and reject) all possible pairs of objects, merely all the pairs within a region. Thus a task that is $O(n^2)$ naively becomes $O(kn)$ where $k$ is the number of objects in a partition.
In consideration of these trade-offs, two-level partitioning seems to be a conceptually simple way to blend the advantages of both extremes. Queries can be fragmented in terms of coarse partitions (“chunks”), and spatial near-neighbor joins can be executed over more fine partitions (“sub-chunks”) within each partition. To avoid the overhead of the sub-chunks for non-join queries, the system can store chunks and generate sub-chunks on-demand for spatial join queries. On-the-fly generation for joins is cost-effective due to the drastic reduction of pairs, which is true as long as there are many sub-chunks for each chunk.

Overlap

A strict partitioning eliminates nearby pairs where objects from adjacent partitions are paired. To produce correct results under strict partitioning, nodes need access to objects from outside partitions, which means that data exchange is required. To avoid this, each partition can be stored with a precomputed amount of overlapping data. This overlapping data does not strictly belong to the partition but is within a preset spatial distance from the partition's borders. Using this data, spatial joins can be computed correctly within the preset distance without needing data from other partitions that may be on other nodes.

Overlap is needed only for the Object Catalog, as all spatial correlations will be run on that catalog only. Guided by the experience from other projects including SDSS, we expect to preset the overlap to ~1 arcmin, which results in duplicating approximately 30% of the Object Catalog.

Spherical geometry

Support for spherical geometry is not common among databases and spherical geometry-based partitioning was non-existent in other solutions when we decided to develop Qserv. Since spherical geometry is the norm in recording positions of celestial objects (right-ascension and declination), any spatial partitioning scheme for astronomical object must account for its complexities.

3.3.7 Technology Choice

As explained in chapter 5., no off-the-shelf solution meets the above requirements today, and RDBMS is a much better fit than Map/Reduce-based system, primarily due to features such as indexes, schema and speed. For that reason, our baseline architecture consists of custom software built on two production components: an open source, “simple”, single-node, non-parallel DBMS (MySQL) and Xrootd [2]. To ease potential future DBMS migrations, the communication with the underlying DBMS relies on basic DBMS functionality only, and avoids any vendor-specific features and additions.
4. Requirements

The key requirements driving the LSST database architecture include: incremental scaling, near-real-time response time for ad-hoc simple user queries, fast turnaround for full-sky scans/correlations, reliability, and low cost, all at multi-petabyte scale. These requirements are primarily driven by the ad-hoc user query access.

4.1 General Requirements

**Incremental scaling.** The system must scale to tens of petabytes and trillions of rows. It must grow as the data grows and as the access requirements grow. New technologies that become available during the life of the system must be able to be incorporated easily. Expected sizes for the largest database catalogs (for the last data release) are captured in the table below. For further storage, disk and network bandwidth and I/O analyses, see [1].
### Table

<table>
<thead>
<tr>
<th>Table</th>
<th>Size [TB]</th>
<th>rows</th>
<th>columns</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>109</td>
<td>~38 billion</td>
<td>~500</td>
<td>Most heavily used, for all common queries on stars/galaxies, including spatial correlations and time series analysis using summarized information</td>
</tr>
<tr>
<td>CalibSource</td>
<td>24</td>
<td>~100 billion</td>
<td>~25</td>
<td>Sources used for calibration</td>
</tr>
<tr>
<td>DiaSource</td>
<td>71</td>
<td>~200 billion</td>
<td>~50</td>
<td>Alert-related follow up analysis</td>
</tr>
<tr>
<td>Source</td>
<td>3,600</td>
<td>~5 trillion</td>
<td>~100</td>
<td>Time series analysis of bright objects and detections</td>
</tr>
<tr>
<td>ForcedSource</td>
<td>1,089</td>
<td>~23 trillion</td>
<td>~7</td>
<td>Specialized analysis of faint objects and detections</td>
</tr>
</tbody>
</table>

**Reliability.** The system must not lose data, and it must provide at least 98% up time in the face of hardware failures, software failures, system maintenance, and upgrades.

**Low cost.** It is essential to not overrun the allocated budget, thus a cost-effective, preferably open-source solution is strongly preferred.

### 4.2 Data Production Related Requirements

In a nutshell, the LSST database catalogs will be generated by a small set of production pipelines:

- **Data Release Production** – it produces all key catalogs. Ingest rates are very modest, as DRP takes several months to complete and is dominated by cpu-intensive application jobs. Ingest can be done separately from pipeline processing, as an post-processing step.
- **Nightly Alert Production** – it produces difference image sources, and inserts/updates the Object and Moving Object catalogs. Since alerts need to be generated in under a minute after data has been taken, data has to be ingested/updated in almost-real time. The number of rows updates/ingested is modest: ~40K new rows and updates every ~30 sec.
- **Calibration Pipeline** – it produces calibration information. Due to small data volume and no stringent timing requirements, ingest bandwidth needs are very modest.

In addition, the camera and telescope configuration is captured in the Engineering & Facility Database. Data volumes are very modest.

Further, the up-to-date catalog will need to be updated every 24h. This catalog should not be taken off-line for extended periods of time.
4.3 Query Access Related Requirements

Produced catalogs need to be available for ad-hoc querying by a wide spectrum of users, including professional astronomers, amateur astronomers and general public. The load from the query access from these users is described as:

1. 300 low-volume users querying against stored meta-data only over 10 million objects in the object catalog or 1 degree$^2$ of the image archive, retrieving a 500 Mbyte dataset from a single cluster/partition, and receiving it at 100Mbps. Query response time should be less than 10 seconds. We have later estimated that 50 out of the 300 users will be active at any given time (will be running a query). That is because users will wait for query result, then will need to think before submitting another query$^1$.

2. 10 high-volume users querying against stored and derived meta-data with moderately complex conditions, over 1 billion objects in the object catalog or the full FOV of the image archive, retrieving a 6 Gbyte dataset from multiple clusters/partitions, and receiving it at 1Gbps. Query response time should be less than 1 hour.

3. 1 super-high-volume user querying against stored and derived meta-data with highly complex conditions and 3-point correlation, over 10 billion objects in the object catalog, retrieving a 10 Gbyte dataset from all clusters/partitions, and receiving it at 10Gbps. Query response time should be less than 10 days.

Query access from pipelines and QA is expected to be significantly less challenging than user query access (for example, a DRP might require few full table scans over the period of several weeks.

**Real time.** A large fraction of ad-hoc user access will involve so called “low-volume” queries – queries that touch small area of sky, or request small number of objects. These queries are required to be answered in under 10 sec. On average, we expect to see ~50 such queries running at any given time.

**Fast turnaround.** High-volume queries – queries that involve full-sky scans are expected to be answered in 1 hour, while more complex full-sky spatial and temporal correlations are expected to be answered in ~10 hours. ~20 simultaneous high-volume queries are expected to be running at any given time.

**Cross-matching.** Occasionally, LSST database catalog will need to be cross-matched with external catalogs: both large, such as SDSS, SKA or GAIA, and small, such as small amateur data sets. Users should be able to save results of their queries, and access them during subsequent queries.

---

$^1$ Discussed/agreed at Database Telecon July 12 2006
**Query complexity.** The system needs to handle complex queries, including spatial correlations, time series comparisons. Spatial correlations are required for the Object catalog only – this is an important observation, as this class of queries requires highly specialized, 2-level partitioning with overlaps.

**Ad-hoc.** It is impossible to predict all types of analysis astronomers will run. The unprecedented volume and scope of data might enable new kind of analysis, and new ways of analysis.

**Flexibility.** Sophisticated end users need to be able to access all this data in a flexible way with as few constraints as possible. We may have to allow the most sophisticated end users to express queries directly in SQL. It is not yet clear whether the full language is necessary or if a subset is adequate, and, if so, what operations need to be part of that subset.

### 4.4 Discussion

The above requirements have important implications on the LSST data access architecture.

- The system must allow rapid selection of small number of rows out of multi-billion row tables. To achieve this, efficient data indexing is essential.
- The system must efficiently join multi-trillion with multi-billion row tables. Denormalizing these tables to avoid common joins, such as Object with Source or Object with ForcedSource, would be prohibitively expensive.
- The system must provide high data bandwidth. In order to process terabytes of data in minutes, data bandwidths on the order of tens to hundreds of gigabytes per second are required.
- To achieve high bandwidths, to enable expandability, and to provide fault tolerance, the system will need to run on a distributed cluster composed of multiple machines.
- The most effective way to provide high-bandwidth access to large amounts of data is to partition the data, allowing multiple machines to work against distinct partitions. Data partitioning is also important to speed up some operations on tables, such as index building.
- Multiple machines and partitioned data in turn imply that at least the largest queries will be executed in parallel, requiring the management and synchronization of multiple tasks.
- Limited budget implies the system needs to get most out available hardware, and scale it incrementally as needed. The system will be disk I/O limited, and therefore we anticipate attaching multiple queries to a single table scan (shared scans) will be a must.
More on query complexity and access patterns

A compilation of representative queries provided by the LSST Science Collaborations, the Science Council, and other surveys have been captured [3.]. These queries can be divided into several distinct groups: analysis of a single object, analysis of objects meeting certain criteria in a region or across entire sky, analysis of objects close to other objects, analysis that require special grouping, time series analysis and cross match with external catalogs. They give hints as to the complexity required: these queries include distance calculations, spatially localized self-joins, and time series analysis.

Small queries are expected to exhibit substantial spatial locality (refer to rows that contain similar spatial coordinates: right ascension and declination). Some kinds of large queries are expected to exhibit a slightly different form of spatial locality: joins will be among rows that have nearby spatial coordinates. Spatial correlations will be executed on the Object table; spatial correlations will not be needed on Source or ForcedSource tables.

Queries related to time series analysis are expected to need to look at the history of observations for a given Object, so the appropriate Source or ForcedSource rows must be easily joined and aggregate functions operating over the list of Sources must be provided.

The query complexity has important implications on the overall architecture of the entire system.

5. Potential Solutions - Research

The two most promising technologies able to scale to LSST size available today are Relational Database Management Systems (RDBMS) and Map/Reduce (MR): the largest DBMS system, based on commercial Teradata and reaching 20+ petabytes is hosted at eBay, and the largest MR-based systems, reaching many tens of petabytes are hosted at Google, Facebook, Yahoo! and others.

5.1 The Research

To decide which technology best fits the LSST requirements, we did an extensive market research and analyses, reviewed relevant literature, and run appropriate stress-tests for selected “promising” candidates, focusing on weak points and the most uncertain aspects. Market research and analyses involved (a) discussions about lessons learned with many industrial and scientific users dealing with large data sets, (b) discussions about existing solutions and future product road-maps with leading solution providers and promising start-ups, and (c), research road-map with leading database researchers from academia. See Appendix E for a list of people and communities we talked to so far.
5.2 The Results

As a result of our research, we determined an RDBMS-based solution involving a shared-nothing parallel database is a much better fit for the LSST needs than MR. The main reasons are availability of indexes which are absolutely essential for low-volume queries and spatial indexes, support for schemas and catalogs, performance and efficient use of resources.

Even though a suitable open source off-the-shelf DMBS capable of meeting LSST needs does not exist today, there is a good chance a system meeting most of the key requirements will be available well before LSST production starts. In particular, there are two very promising by-products of our research:

- we co-initiated, co-founded, and helped bootstrap SciDB – a new open source shared nothing database system, and
- we pushed the development of MonetDB, an open source columnar database into directions very well inlined with LSST needs. We closely collaborate with the MonetDB team – building on our Qserv lessons-learned, the team is trying to add missing features and turn their software into a system capable of supporting LSST needs, demonstration is expected in late 2011.

Both SciDB and MonetDB have strong potential to become the LSST database solution once they mature.

Further, our research led to creation a new, now internationally recognized conference series, Extremely Large Databases (XLDB) [4.], [5.]. As we continue leading the XLDB effort, it gives us a unique opportunity to reach out to a wide range of high-profile organizations dealing with large data sets, and raise awareness of the LSST needs among researchers and developers working on both MR and DBMS solutions.

The remaining of this chapter discusses lessons learned to-date, along with description or relevant tests we run.
5.3 Map/Reduce-based and NoSQL Solutions

Map/Reduce is a software framework to support distributed computing on large data sets on clusters of computers. Google’s implementation [6.], believed to be the most advanced, is proprietary, and in spite of Google being one of the LSST collaborators, we were unable to gain access to any of their MR software or infrastructure. Additional Google-internal MR-related projects include BigTable [7.], Chubby [8.], and Sawzall [9.]. BigTable addresses the need for rapid searches through a specialized index; Chubby adds transactional support; and Sawzall is a procedural language for expressing queries. Most of these solutions attempt to add partial database-like features such as schema catalog, indexes, and transactions. The most recent MR developments at Google are Dremel [10.] - an interactive ad-hoc query system for analysis of read-only data, and Tenzing – a full SQL implementation on the MR Framework [11.].

In parallel to the closed-source systems at Google, similar open-source solutions are built by a community of developers led by Facebook, Yahoo! and Cloudera, and they already gained wide-spread acceptance and support. The open source version of MR, Hadoop, has became popular in particular among industrial users. Other solutions developed on top (and “around”) Hadoop include HBase (equivalent of BigTable), Hive (concept similar to Google's Dremel), Pig Latin (equivalent to Google's Sawzall), Zookeeper (equivalent to Google's Chubby), Simon, and others. As in Google's case, the primary purpose of building these solutions is adding database-features on top of MR. Hadoop is commercially supported by Cloudera and Hortonworks [12., 13.]

We have experimented with Hadoop (0.20.2) and Hive (0.7.0) in mid 2010 using a 1 billion row USNO-B data set on a 64 node cluster [14.]. Common LSST queries were tested, ranging from low-volume type (such as finding a single object, selecting objects near other known object), through high-volume ones (full table scans) to complex queries involving joins (join was implemented in a standard way, in the reduce step). The results were discussed with Hadoop/Hive experts from Cloudera.

Independently, Microsoft developed a system called Dryad, geared towards executing distributed computations beyond “flat” Map and Reduce, along with a corresponding language called LINQ. Due to its strong dependence of Windows OS and limited availability, use of Dryad outside of Microsoft is very limited.

Further, there is a group of new emerging solutions often called as NoSQL. The two most popular ones are Cassandra, and MongoDB.

The remaining of this section discusses all of the above-mentioned products.

Further details about individual MR and no-SQL solutions can be found in Appendix A and B.

---

2 Through our XLDB efforts, Google has provided us with a preprint of a Tenzing manuscript accepted for publication at VLDB 2011.
5.4 DBMS Solutions

Database systems have been around for much longer than MR, and therefore they are much more mature. They can be divided into many types: parallel/single node, relational/object-oriented, columnar/row-based, some are built as appliances. Details about individual DBMS products and solutions we considered and/or evaluated can be found in Appendix B.

5.4.1 Parallel DBMSes

Parallel databases, also called MPP DBMS (massively parallel processing DBMS), improve performance through parallelization of queries: using multiple CPUs, disks and servers in parallel. Data is processed in parallel, and aggregated into final result. The aggregation may include computing average, max/min and other aggregate functions. This process is often called scatter-gather, and it is somewhat similar to map and reduce stages in the MR systems.

Shared-nothing parallel databases, which fragment data and in many cases use an internal communications strategy similar to MR, scale significantly better than single-node or shared-disk databases. Teradata uses proprietary hardware, but there are a number of efforts to leverage increasingly-fast commodity networks to achieve the same performance at much lower cost, including Greenplum, DB2 Parallel Edition, Aster Data, GridSQL, ParAccel, InfiniDB, SciDB, and Project Madison at Microsoft (based on DATAllegro, acquired by Microsoft in 2008). Most of these efforts are relatively new, and thus the products are relatively immature. EBay's installation used to be based on Greenplum in 2009 and reached 6.5 PB, and now it is approaching 30 PB an is based on Teradata's Singularity. Some of these databases have partition-wise join, which can allow entity/observation join queries to execute more efficiently, but none allow overlapping partitions, limiting the potential performance of pairwise analysis.

Microsoft SQL Server offers Distributed Partitioned Views, which provide much of the functionality of a shared-nothing parallel database by federating multiple tables across multiple servers into a single view. This technology is used in the interesting GrayWulf project [15., 16.], which is designed to host observational data consisting of Pan-STARRS PS1 [17.] astronomical detections and summary information about the objects that produced them. GrayWulf partitions observation data across nodes by “zones” [18.], but these partitions cannot overlap. Fault tolerance is built in by having three copies of the data, with one undergoing updates – primarily appending new detections – and the other two in a hot/warm relationship for failover. GrayWulf has significant limitations, however. The object information for the Pan-STARRS PS1 data set is small enough (few TB) that it can be materialized on a single node. The lack of partition-wise join penalizes entity/observation join queries and pairwise analysis. The overall system design is closely tied to the commercial SQL Server product, and re-hosting it on another RDBMS, in particular an open source one, would be quite difficult.
The MPP database is ideal for the LSST database architecture. Unfortunately, the only scalable, proven off-the-shelf solutions are commercial and expensive: Teradata, Greenplum. Both systems are (or recently were) behind today world's largest production database systems at places such as eBay [19., 20.] and Walmart [21.]. IBM's DB2 “parallel edition”, even though it implements a shared-nothing architecture since mid-1990 focuses primarily on supporting unstructured data (XML), not large scale analytics.

The emergence of several new startups, such as Aster Data, DataAllegro, ParAccel, GridSQL and SciDB is promising, although some of them have already been purchased by the big and expensive commercial RDBMSes: Teradata purchased Aster Data, Microsoft purchased DataAllegro. To date, the only shared-nothing parallel RDBMS available as open source is SciDB – its first production version (v11.06) was released in June 2011. ParAccel is proprietary and still in start-up mode. After testing GridSQL we determined it does not offer enough benefits to justify using it, the main cons include limited choices of partitioning types (hash partitioning only), lack of provisions for efficient near neighbor joins, poor stability and lack of good documentation.

SciDB is the only parallel open source DBMS currently available on the market. It is a columnar, shared-nothing store based on an array data model. The project has been inspired by the LSST needs [22.], and the LSST Database team is continuously in close communication with the SciDB developers. SciDB’s architectural features of chunking large arrays into overlapping chunks and distributing these chunks across a shared nothing cluster of machines match the LSST database architecture. Initial tests run with the v0.5 SciDB release exposed architectural issues with SciDB essential for LSST, related to clustering and indexing multi-billion, sparse arrays of objects in a 2-dimensional (ra, declination) space. These issues are expected to be addressed in the recently released v11.06, we are currently re-running the tests and continuing our evaluation of SciDB.

5.4.2 Object-oriented solutions

The object-oriented database market is very small, and the choices are limited to a few small proprietary solutions, including Objectivity/DB and InterSystems Caché. Objectivity/DB was used by the BaBar experiment in 1999 – 2002, and the BaBar database reached a petabyte [23.]. The members of LSST database team, being the former members of the BaBar database team are intimately familiar with the BaBar database architecture. The Objectivity/DB was used primarily as a simple data store, all the complexity, including custom indices had to be all built in custom, user code. Given that, combining with the challenges related to porting and debugging commercial system led as to a conclusion Objectivity/DB is not the right choice for LSST.
InterSystems Caché has recently been chosen as the underlying system for the European Gaia project [24., 25.], however so far the Gaia project focused primarily on ingest-related aspects of the system, and did not have a chance to research analytical capabilities of Caché at scale. We have been in contact with both Gaia database team and the Caché representatives, and are in the process of establishing a closer collaboration to determine whether Caché might be a good fit for LSST database challenges.

5.4.3 Row-based vs columnar stores

Row-based stores organize data on disk as rows, while columnar store – as columns. Column-store databases emerged relatively recently, and are based on the C-store work [26.]. By operating on columns rather than rows, they are able to retrieve only the columns required for a query and greatly compress the data within each column. Both reduce disk I/O and hence required hardware by a significant factor for many analytical queries on observational data that only use a fraction of the available columns. Current column stores also allow data to be partitioned across multiple nodes, operating in a shared-nothing manner. Column stores are less efficient for queries retrieving small sets of full-width records, as they must reassemble values from all of the columns.

Our baseline architecture assumes all large-volume queries will be answered through shares scans, which reduces wasting disk I/O for row-based stores: multiple queries attached the the same scan will typically access most columns (collectively).

Work done at Google (using Dremel) has claimed that “the crossover point often lies at dozens of fields but it varies across data sets” [10.]. In our case, most frequently accessed table: Object, will have about “30 dozens” columns. Source table will have ~10 dozens, and ForcedSource less than one, however it is almost guaranteed all ForcedSource columns will be always accessed. Low query selectivity (expected to be <1% for full table scans) combined with late materialization (postponing row assembly until the last possible moment) is expected to further boost effectiveness of columnar stores.
The two leading row-based DBMSes are MySQL and PostgreSQL. Of these two, MySQL is better supported, and has much wider community of users, although both are commercially supported (MySQL: Oracle, PostgreSQL: EnterpriseDB, Greenplum). PostgreSQL tends to focus more on OLTP, while MySQL is closer to our analytical needs, although both are weak in the area of scalability. One of the strongest points of PostgreSQL is spatial GIS support, however MySQL has recently started rewriting their GIS modules. After Oracle's acquisition of MySQL (through Sun), several MySQL forks of MySQL code appeared (Monte Program, Percona, Drizzle). All except Drizzle can be to the large extent treated as “the same” software – the difference between them is relatively small. Drizzle stands out, it is a completely re-worked version. It focuses on OLTP more than MySQL, in particular, it does not support the MyISAM non-transactional engine, which is the engine of our choice for the LSST query access load, due to no storage overheads and good performance. Once the MariaDB engine becomes available in Drizzle, Drizzle might become a viable replacement of MySQL for LSST.

Many commercial row-bases DBMSes exist, including Oracle, SQL Server, DB2, however given they are (1) expensive and (2) they are single node (not MPP), they do not fit well into LSST needs.

Columnar stores are starting to gain on popularity. Although the list is already relatively large, the number of choices worth considering is relatively small. Today's most popular commercial choice is HP Vertica, and the open source solutions include MonetDB and Calpont's InfiniDB. The latter also implements shared nothing MPP, however multi-server version is only available as part of the commercial edition.

With help from Calpont, we evaluated InfiniDB and demonstrated it could be used for the LSST system – we run the most complex (near neighbor) query. Details are available in Appendix C.

We are now working closely with the MonetDB team, including the main architect of the system, Martin Kersten and two of his students who worked on porting MonetDB to meet LOFAR database needs. The MonetDB team has run some basic tests using astronomical data (USNOB as well as our DC3b-PT1.1 data set). During the course of testing our common queries they implemented missing features such as support for user defined functions, and are actively working on further extending MonetDB to build remaining missing functionality, in particular ability to run as a shared-nothing system. To achieve that, existing MonetDB server (merovingian) has to be extended. Table partitioning and overlaps (on a single node) can be achieved through table views, although scalability to LSST sizes still need to be tested. Cross-node partitioning requires new techniques, and the MonetDB team is actively working on it. A demonstration of MonetDB system capable of supporting LSST needs is expected in late 2011.

---

3 Based on discussions with MySQL developers at MySQL User Conference April 2011
5.4.4 Appliances

Appliances rely on specialized hardware to achieve performance. In general, we are skeptical about appliances, primarily because they are locking us into this specialized hardware. In addition, appliances are usually fast, however their replacement cost is high, so often commodity hardware is able to catch up, or even exceed the performance of an appliance after few years (the upgrade of an appliance to a latest version is usually very costly).

5.5 Comparison and Discussion

The MR processing paradigm became extremely popular in the last few years, in particular among peta-scale industrial users. Most industrial users with peta-scale data sets heavily rely on it, including places such as Google, Yahoo!, Amazon or Facebook, and even eBay has recently started using Hadoop for some of their (offline, batch) analysis. The largest (peta-scale) RDBMS-based systems all rely on shared-nothing, MPP technology, and almost all on expensive Teradata solutions (eBay, Walmart, Nokia, for few years eBay used Greenplum but they switched back to Teradata's Singularity).

In contrast, science widely adopted neither RDBMS nor MR. The community with largest data set: HEP is relying on home-grown system, augmented by a DBMS (typically Oracle or MySQL) for managing the metadata. This is true for most HEP experiments of the last decade (with the exception of BaBar which initially used Objectivity), as well as the LHC experiments. In astronomy, most existing systems as well as the systems starting in the near future are RDBMS-based (SDSS – SQL Server, Pan-STARRS – SQL Server, 2MASS – Informix, DES – Oracle, LOFAR – MonetDB, Gaia – Cache). It is worth noting that none of these systems was large enough so far to break a single-node barrier, with the exception of Pan-STARRS. Geoscience relies primarily on netCDF/HDF5 files with metadata in a DBMS. Similar approach is taken by bio communities we have talked to. In general, MR approach has not been popular among scientific users so far.

The next few sections outline key differences, strengths and weaknesses of MR and RDBMS, and the convergence.

APIs
In the MR world, data is accessed by a pair of functions, one that is “mapped” to all inputs, and one that “reduces” the results from the parallel invocations of the first. Problems can be broken down into a sequence of MR stages whose parallel components are explicit. In contrast, a DBMS forces programmers into less natural, declarative thinking, giving them very little control over the flow of the query execution; this issue might partly go away by interacting with database through a user defined function (UDFs), which are becoming increasingly popular. They must trust the query optimizer's prowess in “magically” transforming the query into a query plan. Compounding the difficulty is the optimizer's unpredictability: even one small change to a query can make its execution plan efficient or painfully slow.

The simplicity of the MR approach has both advantages and disadvantages. Often a DBMS is able to perform required processing on the data in a small number of passes (full table scans). The limited MR operators on the other hand may lead to many more passes through the data, which requires more disk I/O thus reduces performance and increases hardware needed. Also, MR forced users to code a lot of operations typically provided by an RDBMS by-hand – these include joins, custom indexes or even schemas.

**Scalability, fault tolerance and performance**

The simple processing framework of MR allows to easily, incrementally scale the system out by adding more nodes as needed. Frequent check-pointing done by MR (after every “map” and every “reduce” step) simplifies recoverability, at the expense of performance. In contrast, databases are built with the optimistic assumptions that failures are rare: they generally checkpoint only when necessary. This has been shown through various studies [28].

The frequent checkpointing employed by MR, in combination with limited set of operators discussed earlier often leads to inefficient usages of resources in MR based systems. Again, this has been shown through various studies. EBay's case seems to support this as well: back in 2009 when they managed 6.5 petabytes of production data in an RDBMS-based system they relied on a mere 96 nodes, and based on discussions with the original architects of the eBay system, to achieve comparable processing power through MR, many thousand nodes would be required.

**Flexibility**

MR paradigm treats a data set as a set of key-value pairs. It is structure-agnostic, leaving interpretation to user code and thus handling both poorly-structured and highly-complex data. Loose constraints on data allow users to get to data quicker, bypassing schema modeling, complicated performance tuning, and database administrators. In contrast, data in databases are structured strictly in records according to well-defined schemata.
While adjusting schema with ease is very appealing, in large scientific projects like LSST, schema has to be carefully thought through to meet needs of many scientific collaborations, each having different set of requirements. The flexibility would be helpful during designing/debugging, however it is of a lesser value for science, comparing to industry with rapidly changing requirements, and strong focus on agility.

**Cost**

As of now, the most popular MR framework, *Hadoop*, is freely available as open source. In contrast, none of the freely available RDBMSes implements a shared-nothing MPP DBMS (to date), with the exception of still-immature SciDB.

**Supporting data correlations**

MR is perfect for processing large data sets in parallel for as long as the processed data set can be easily divided into independent subsets, and each subset can be processed independently. This approach works particularly well on uncorrelated and often unstructured data sets, such as the content of HTML webpages or user profiles. Segments or chunks of data are typically randomly distributed (hashed) across the available nodes and processed in parallel. This model is significantly less well-suited for pairwise spatial analysis such as these required by astronomers. In astronomy\(^4\), the property of data *adjacency* is very important: astronomical data is highly correlated, in both spatial and temporal dimensions, and data is very frequently queried in a way that takes spacial adjacency into account. These queries include near neighbor searches (“find things near other things” type of queries) and density based queries (“find things in dense/sparse regions” type of queries).

Databases often offer spatial extensions and spatial indices to facilitate rapid location of objects in a two-dimensional space, as well as spatially correlations. While this helps, some extra measures need to be taken even in index-rich RDBMS to enable executing near neighbor type queries on multi-billion row tables. These measures, such as implementing overlapping partitions and co-locating adjacent partitions are easier to implement on top of an RDBMS than MR, primarily because MR gives user no control over where partitions are physically located.

---

\(^4\) As well as in most other sciences dealing with images, including geo-science and medical imaging.
From the LSST perspective, plain MR does not meet project's need, in particular the low-volume query short response time needs. Significant effort would be required to alleviate Hadoop's high latency (today's solution is to run idle MR daemons, and attach jobs to them, which pushes the complexity of starting/stopping jobs onto user code). Also, table joins, typically done in reduce stage, would have to be implemented as maps to avoid bringing data for joined tables toReducer – in practice this would require implementing a clever data partitioning scheme. The main advantages of using MR as a base technology for the LSST system include scalability and fault-tolerance, although as alluded above, these features come at a high price: inefficient use of resources (full checkpointing between each Map and each Reduce step), and triple redundancy.

**Summary**

The key features of an ideal system, along with the comments for both Map/Reduce and RDBMS are given in the table below.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Map/Reduce</th>
<th>RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared nothing, MPP, columnar</td>
<td>Implements it.</td>
<td>Some implement it, but only as commercial, non open source to date, except not-yet-mature SciDB.</td>
</tr>
<tr>
<td>Overlapping partitions, needed primarily for near-neighbor queries</td>
<td>Nobody implements this.</td>
<td>Only SciDB implements this to-date.</td>
</tr>
<tr>
<td>Shared scans (primarily for complex queries that crunch through large sets of data)</td>
<td>This kind of logic would have to be implemented by us.</td>
<td>There is a lot of research about shared scans in databases. Implemented by Teradata. Some vendors, including SciDB are considering implementing it.</td>
</tr>
<tr>
<td>Efficient use of resources</td>
<td>Very inefficient.</td>
<td>Much better than MR.</td>
</tr>
<tr>
<td>Catalog/schema</td>
<td>Started adding support, e.g., Hive.</td>
<td>Much better than in MR.</td>
</tr>
<tr>
<td>Indexes (primarily for simple queries from public that require real time response)</td>
<td>Started adding support, e.g., Hive.</td>
<td>Much better than in MR.</td>
</tr>
</tbody>
</table>
Open source | Hadoop (although it is implemented in Java, not ideal from LSST point of view) | No shared-nothing MPP available as open source yet except still-immature SciDB. We expect there will be several by the time LSST needs it (SciDB, MonetDB, ParAccel and others)

Convergence

Despite their differences, the database and MR communities are learning from each other and seem to be converging.

The MR community has recognized that their system lacks built-in operators. Although nearly anything can be implemented in successive MR stages, there may be more efficient methods, and those methods do not need to be reinvented constantly. MR developers have also explored the addition of indexes, schemas, and other database-ish features. Some have even built a complete relational database system on top of MR.

The database community has benefited from MR's experience in two ways:

1. Every parallel shared-nothing DBMS can use the MR execution style for internal processing – while often including more-efficient execution plans for certain types of queries. Though systems such as Teradata or IBM's DB2 Parallel Edition have long supported this, a number of other vendors are building new shared-nothing-type systems. It is worth noting that these databases typically use MR-style execution for aggregation queries.

2. Databases such as Greenplum (part of EMC) and Aster Data (part of Teradata since March 2011) have begun to explicitly support the MR programming model with user-defined functions. DBMS experts have noted that supplying the MR programming model on top of an existing parallel flow engine is easy, but developing an efficient parallel flow engine is very hard. Hence it is easier for the DMBS community to build map/reduce than for the map/reduce community to add full DBMS functionality.

The fact MR community is rapidly adding database/SQL like features on top of their plain MR (Tenzing, Hive, HadoopDB, etc), confirms the need for database-like features (indexes, schemas, catalogs, sql).

---

5 An example of that is Hive: http://hadoop.apache.org/hive/

6 An example of that is HadoopDB. http://db.cs.yale.edu/hadoopdb/hadoopdb.html

7 ParAccel, Vertica, Aster Data, Greenplum, DATAllegro (now part of Microsoft), Dataupia, Exasol and SciDB
As we continue monitoring the latest development in both RDBMS and MR communities and run more tests, we expect to re-evaluate our choices as new options become available.

6. Design Trade-offs
The LSST database design involves many architectural choices. Example of architectural decisions we faced include how to partition the tables, how many levels of partitioning is needed, where to use an index, how to normalize the tables, or how to support joins of the largest tables. This chapters covers the test we run to determine the optimal architecture of MySQL-based system.

6.1 Standalone Tests

6.1.1 Spatial join performance
This test was run to determine how quickly we can do a spatial self-join (find objects within certain spatial distance of other objects) inside a single table. Ultimately, in our architecture, a single table represents a single partition (or sup-partition). The test involved trying various options and optimizations such as using different indexes (clustered and non clustered), precalculating various values (like COS(RADIANS(decl))), and reordering predicates. We run these tests for all reasonable table sizes (using MySQL and PostgreSQL). We measured CPU and disk I/O to estimate impact on hardware. In addition, we re-run these tests on the lsst10 machine at NCSA to understand what performance we may expect there for DC3b. These tests are documented at [http://dev.lsstcorp.org/trac/wiki/dbSpatialJoinPerf](http://dev.lsstcorp.org/trac/wiki/dbSpatialJoinPerf).

6.1.2 Building sub-partitions
Based on the “spatial join performance” test we determined that in order to speed up self-joins within individual tables (partitions), these partitions need to be very small, \( O(\text{few } K) \) rows. However, if we partition large tables into a very large number of small tables, this will result in unmanageable number of tables (files). So, we determined we need a second level of partitioning, which we call sub-partition on the fly. This test included:

- sub-partitioning through queries:
  1. one query to generate one sub-partition
  2. relying on specially introduced column (subPartitionId).
- segregating data into sub-partitions in a client C++ program, including using a binary protocol.

We timed these tests. This test is described at [http://dev.lsstcorp.org/trac/wiki/dbBuildSubPart](http://dev.lsstcorp.org/trac/wiki/dbBuildSubPart).
6.1.3 Sub-partition overhead

We also run detailed tests to determine overhead of introducing sub-partitions. For this test we used a 1 million row table, measured cost of a full table scan of such table, and compared it against scanning through a corresponding data set partitioned into sub-partitioned. The tests involved comparing in-memory with disk-based tables. We also tested the influence of introducing “skinny” tables, as well as running sub-partitioning in a client C++ program, and inside a stored procedure. These tests are described at http://dev.lsstcorp.org/trac/wiki/dbSubPartOverhead

6.1.4 Avoiding materializing sub-partitions

We tried to run near neighbor query on a 1 million row table. A starting point is 1000 sec which is ~16 min 40 sec (based on earlier tests we determined it takes 1 sec to do near neighbor for 1K row table).

The testing included:

• Running near neighbor query by selecting rows with given subChunkId into in memory table and running near neighbor query there. It took 7 min 43 sec.
• Running near neighbor query by running neighbor once for each subChunkId, without building sub-chunks. It took 39 min 29 sec.
• Running near neighbor query by mini-near neighbor once for each subChunkId, without building sub-chunks, using in-memory table. It took 13 min 13 sec.

6.1.5 Billion row table / reference catalog

One of the catalogs we will need to support is the reference catalog, even in DC3b it is expected to contain about one billion rows. We have run tests with a table containing 1 billion rows catalog (containing USNO-B data) to determine how feasible it is to manage a billion row table without partitioning it. These tests are described in details at: http://dev.lsstcorp.org/trac/wiki/DbStoringRefCat

6.1.6 Compression

We have done extensive tests to determine whether it is cost effective to compress LSST databases. This included measuring how different data types and indexes compress, and performance of compressing and decompressing data. These tests are described in details at http://dev.lsstcorp.org/trac/wiki/compression

6.1.7 Full table scan performance

To determine performance of full table scan, we measured:
1. raw disk speed with “\textit{dd if=<large file> of=/dev/zero}” and got 54.7 MB/sec (2,048,000,000 bytes read in 35.71 sec)
2. speed of “\textit{select count(*) from XX where muRA = 4.3}” using a 1 billion row table. There was no index on muRA, so this forced a full table scan. Note that we did not do “SELECT *” to avoid measuring speed of converting attributes. The scan of 72,117,127,716 bytes took 28:49.82 sec, which is 39.8 MB/sec.

So, based on this test the full table scan can be done at \textit{73\% of the raw disk speed} (using MySQL MyISAM).

\textbf{6.1.8 Multi-node partitioning overheads}

Based on preliminary tests we see \textasciitilde 4 sec overheads. This section will be expanded and details will be given soon. Note that our system is completely unoptimized and it is likely a lot of easy to implement optimizations are possible.

We have not yet measured overheads related to combining results from multiple nodes. We expect these to be within few sec.

\textbf{6.1.9 Low-volume queries}

A typical low-volume queries to the best of our knowledge can be divided into two types:

- analysis of a single object. This typically involves locating a small number of objects (typically just one) with given objectIds, for example find object with given id, select attributes of a given galaxy, extract time series for a given star, or select variable objects near known galaxy. Corresponding representative queries:
  \begin{verbatim}
  SELECT * from Object where objectId=<xx>
  SELECT * from Source where objectId =<xx>
  \end{verbatim}

- analysis of objects meeting certain criteria in a small spatial region. This can be represented by a query that selects objects in a given small ra/dec bounding box, so e.g.:
  \begin{verbatim}
  SELECT * FROM Object
  WHERE ra BETWEEN :raMin AND :raMax
  AND decl BETWEEN :declMin AND :declMax
  AND zMag BETWEEN :zMin AND :zMax
  \end{verbatim}

Each such query will typically touch one or a few partitions (few if the needed area is near partition edge). In this test we measured speed for a single partition.
Proposed partitioning scheme will involve partitioning each large table into a “reasonable” number of partitions, typically measured in low tens of thousands. Details analysis are done in the storage spreadsheet (Docushare LDM-141). Should we need to, we can partition the largest tables into larger number of smaller partitions, which would reduce partition size. Given the hardware available and our time constraints, so far we have run tests with up to 10 million row partition size.

We determined that if we use our custom spatial index (“subChunkId”), we can extract 10K rows out of a 10 million row table in 30 sec. This is too long – low volume queries require under 10 sec response time. However, if we re-sort the table based on our spatial index, that same query will finish in under 0.33 sec.

We expect to have 50 low volume queries running at any given time. Based on details disk I/O estimates, we expect to have ~200 disk spindles available in DR1, many more later. Thus, it is likely majority of low volume queries will end up having a dedicated disk spindle, and for these that will end up sharing the same disk, caching will likely help.

Note that these tests were done on fairly old hardware (7 year old).

In summary, we demonstrated low-volume queries can be answered through an index (objectId or spatial) in well under 10 sec.

6.1.10 Solid state disks

We also run a series of tests with solid state disks to determine where it would be most cost-efficient to use solid state disks. The tests are described in details in [29.]  

6.2 Data Challenge Related Tests

During each data challenge we test some aspects of database performance and/or scalability. In DC1 we demonstrated ingest into database at the level of 10% of DR1, in DC2 we demonstrated near-real-time object association, DC3 is demonstrating catalog construction and DC4 will demonstrate the end user query/L3 data production.

In addition to DC-related tests, we are running standalone tests, described in details in chapter 9.
6.2.1 **DC1: data ingest**

We run detailed tests to determine data ingest performance. The test included comparing ingest speed of MySQL against SQL Server speed, and testing different ways of inserting data to MySQL, including direct ingest through INSERT INTO query, loading data from ASCII csv files. In both cases we tried different storage engines, including MyISAM and InnoDB. Through these tests we determined the overhead introduced by MySQL is small (acceptable). Building indexes for large tables is slow, and requires making a full copy of the involved table. These tests are described in details in Docushare Document-1386.

6.2.2 **DC2: source/object association**

One of the requirements is to associated DiaSource with Object is almost real-time. Detailed study how to achieve that has been done in conjunction with the Data Challenge 2. The details are covered at: [http://lsstdev.ncsa.uiuc.edu/trac/wiki/DC2DbPartitioningTests](http://lsstdev.ncsa.uiuc.edu/trac/wiki/DC2DbPartitioningTests) and the pages linked from there.

6.2.3 **DC3: catalog construction**

In DC3 we demonstrated catalog creation as part of the Data Release Production.

6.2.4 **DC4: end user query/L3 data production**

In DC4 we expect to demonstrate creating/managing Level-3 data products, as well as end-user query access.

7. **Risk Analysis**

7.1 **Potential Key Risks**

Insufficient [database performance and scalability](#) is one of the major risks [30.].

We have a prototype system (Qserv) that will be turned into a production system. Given that a large fraction of its functionality is derived from two stable, production quality, open source components (MySQL and Xrootd), turning it into production system is possible during the LSST construction phase.
A viable alternative might be to use an off-the-shelf system. In fact, an off-the-shelf solution could present significant support cost advantages over a production-ready Qserv, especially if it is a system well supported by a large user and developer community. It is likely that an open source, scalable solution will be available on the time scale needed by LSST (for the beginning of LSST construction a stable beta would suffice, beginning of production scalability approaching few hundred terabytes would be sufficient). Database systems larger than the largest single LSST data set have been successfully demonstrated in production today. For example, eBay manages a 10+ petabyte production database[20.] and expects to deploy a 36 petabyte system later in 2011. For comparison, the largest single LSST data set, including all indexes and overheads is expected to be below 10 petabytes in size, and will be produced ~20 years from now (the last Data Release)\(^8\). The eBay system is based on an expensive commercial DBMS (Teradata), but there is a growing demand for large scale systems and growing competition in that area (Hadoop, SciDB, Greenplum, InfiniDB, MonetDB, Caché and others).

Finally, a third alternative would be to use a closed-source, non free software, such as Caché, InfiniDB or Greenplum (Teradata is too expensive). Some of these systems, in particular Caché and InfiniDB are very reasonably priced.

Potential **problems with off-the-shelf database software** used, such as MySQL is another potential risk. MySQL has recently been purchased by Oracle, leading to doubts as to whether the MySQL project will be sufficiently supported in the long-term. Since the purchase, several independent forks of MySQL software have emerged, including MariaDB (supported by one of the MySQL founders), Drizzle (supported by key architects of MySQL), and Percona. Should MySQL disappear, these open-source, MySQL-compatible\(^9\) systems are a solid alternative. Should we need to migrate to a different DBMS, we have taken multiple measures to minimize the impact:

- our schema does not contain any MySQL specific elements and we have successfully demonstrating using it in other systems such as MonetDB and Microsoft's SQL Server;
- we do not rely on any MySQL specific extensions, with the exception of MySQL Proxy, which can be made to work with non-MySQL systems if needed;
- we minimize the use of stored functions and stored procedures which tend to be DBMS-specific, and instead use user defined functions, which are easier to port (only the interface binding part needs to be migrated).

---

\(^8\) The numbers, both for eBay and LSST are for compressed data sets.

\(^9\) With the exception of Drizzle, which introduced major changes to the architecture.
**Complex data analysis.** The most complex analysis we identified so far include spatial and temporal correlations which exhibit $O(n^2)$ performance characteristics, searching for anomalies and rare events, as well as searching for unknown are a risk as well – in most cases industrial users deal with much simpler, well defined access patterns. Also, some analysis will be ad-hoc, and access patterns might be different than these we are anticipating. Recently, large-scale industrial users started to express strong need for similar types of analyses; understanding and correlating user behavior (time-series of user clicks) run by web companies, searching for abnormal user behavior to detect fraud activities run by banks and web companies, analyzing genome sequencing data run by biotech companies, and what-if market analysis run by financial companies are just a few examples. Typically these analysis are ad-hoc and involve searching for unknowns, similar to scientific analyses. As the demand (by rich, industrial users) for this type of complex analyses grows, the solution providers are rapidly starting to add needed features into their systems.

### 7.2 Risks Mitigations

To mitigate the insufficient performance/scalability risk, we developed Qserv, and demonstrated scalability and performance. In addition, to increase chances an equivalent open-source, community supported, off-the-shelf database system becomes available in the next few years, we initiated the SciDB array-based scientific database project and work closely with its development team. We also closely collaborate with the MonetDB open source columnar database team – building on our Qserv lessons-learned, they are trying to add missing features and turn their software into a system capable of supporting LSST needs. A demonstration is expected in late 2011. Further, to stay current with the state-of-the-art in peta-scale data management and analysis, we continue a dialog with all relevant solution providers, both DBMS and Map/Reduce, as well as with data-intensive users, both industrial and scientific, through the XLDB conference and workshop series we lead, and beyond.

To understand query complexity and expected access patterns, we are working with LSST Science Collaborations and the LSST Science Council to understand the expected query load and query complexity. We have compiled a set of common queries [3.] and distilled this set into a smaller set of representative queries we use for various scalability tests—this set represents each major query type, ranging from trivial low volume, to complex correlations. [31.]. We have also talked to scientists and database developers from other astronomical surveys, including SDSS, 2MASS, Gaia, DES, LOFAR and Pan-STARRS.

To deal with unpredictability of analysis, we will use shared scans. With shared scans, users will have access to all the data, all the columns, even these very infrequently used, at a predictable cost – with shared scans increasing complexity does not increase the expensive disk I/O needs, it only increases the CPU needs.
To keep query load under control, we will employ throttling to limit individual query loads.

8. Implementation of the Query Access Prototype (Qserv)

To demonstrate feasibility of running LSST queries without relying on expensive commercial solutions, and to mitigate risks of not having an off-the-shelf system in time for LSST construction, we built a prototype system for user query access, called Query Service (Qserv). The system relies on two production-quality components: MySQL and Xrootd. The prototype closely follows the LSST baseline database architecture described in chapter 3.

8.1 Components

8.1.1 MySQL

MySQL is used as an underlying SQL execution engine. To control the scope of effort, Qserv uses an existing SQL engine, MySQL, to perform as much query processing as possible. MySQL is a good choice because of its active development community, mature implementation, wide client software support, simple installation, lightweight execution, and low data overhead. MySQL's large development and user community means that expertise is relatively common, which could be important during Qserv's development or long-term maintenance in the years ahead. MySQL's MyISAM storage engine is also lightweight and well-understood, giving predictable I/O access patterns without an advanced storage layout that may demand more capacity, bandwidth, and IOPS from a tightly constrained hardware budget.

It is worth noting, however, that Qserv's design and implementation do not depend on specifics of MySQL beyond glue code facilitating results transmission. Loose coupling is maintained in order to allow the system to leverage a more advanced or more suitable database engine in the future.

8.1.2 Xrootd

The Scalla/Xrootd distributed file system is used to provide a distributed, data-addressed, replicated, fault-tolerant communication facility to Qserv. Re-implementing these features would have been non-trivial, so we wanted to leverage an existing system. Xrootd has provided scalability, fault-tolerance, performance, and efficiency for several years of in the high-energy physics community, and its relatively flexible API enabled its use as a more general communication medium instead of a file system. Since it was designed to serve large data sets, we were confident that it could mediate not only query dispatch communication, but also bulk transfer of results.
A Scalla/Xrootd cluster is implemented as a set of data servers and a redirector(s). A client connects to the redirector, which acts as a caching namespace lookup service that redirects clients to appropriate data servers. In Qserv, Xrootd data servers become Qserv workers by plugging custom code into Xrootd as a custom file system implementation. The Qserv master dispatches work to workers by writing to partition-addressed Xrootd paths and reads results from hash-addressed Xrootd paths.

8.2 Partitioning

In Qserv, large spatial tables are fragmented into spatial pieces in the two-level partitioning scheme. The partitioning space is a spherical space defined by two angles $\phi$ (right ascension/\(\alpha\)) and $\theta$ (declination/\(\delta\)). For example, the Object table is fragmented spatially, using a coordinate pair specified in two columns--right-ascension and declination. On worker nodes, these fragments are represented as tables named Object CC and Object CC SS where CC is the “chunk id” (first-level fragment) and SS is the “sub-chunk id” (second-level fragment of the first larger fragment. Sub-chunk tables are built on-the-fly to optimize performance of spatial join queries. Large tables are partitioned on the same spatial boundaries where possible to enable joining between them.
8.3 Query Generation

Partitioning is hidden from the user, so Qserv rewrites user queries for execution on chunk and sub-chunk tables on worker nodes. We have extended Lubos Vnuk's Sql2SQL grammar to handle the necessary query token and phrase detection to extract characteristics necessary for generating “chunk queries” for dispatch. We have not implemented code for all SQL syntax since doing so is similar in complexity to a complete SQL query execution engine, and SQL is not a simple language.

In Qserv, query parsing serves 5 main functions:

• Detect spatial restrictions. Queries that include spatial restriction do not need to be dispatched on all chunks. This prevents spatial queries from becoming full-sky queries and saves significant worker load as well as overhead for dispatch and management.

• Detect index opportunities. While only one column is indexed in our case, indexing is crucial for optimizing an important class of queries.

• Detect database and table references. Each reference is detected and instrumented so that they may be rewritten. Not all tables are partitioned, and database references are sometimes rewritten as well. Detection also facilitates access restriction.

• Detect aliases and joins. SQL aliases are common, especially in join syntaxes and must be appropriately managed during rewriting.

• Other preparation for results merging and aggregation.

Example

Consider a user query:

```
SELECT AVG(uFlux_SG)
FROM Object
WHERE qserv_areaspec_box(0.0, 0.0, 10.0, 10.0)
AND uRadius_PS > 0.04;
```

The AVG(uFlux_SG) function call is converted into a SUM(uFlux_SG) and COUNT(uFlux_SG) pair for chunk queries and SUM(`SUM(uFlux_SG)`) / SUM(`COUNT(uFlux_SG)`) to aggregate the resulting rows after results from all chunks have been gathered.

The reference to the Object table is converted to LSST.Object_CC, where CC is substituted appropriately for each chunk. The “LSST.” database qualifier is added from the user database context and is necessary for the query to operate in the different context available on worker nodes.
The qserv_areaspec_box(0.0, 0.0, 10.0, 10.0) is used to select a set of chunks over $0.0 < \phi < 10.0$ and $0.0 < \theta < 10.0$, and is rewritten to operate using a user-defined function installed on worker database instances. The Object table is partitioned where $(\phi, \theta)$ are (ra_PS and decl_PS) so the call is rewritten as qserv_ptInSphericalBox(ra_PS, decl_PS,0.0, 0.0, 10.0, 10.0) = 1.

Qserv does not currently support SQL sub-queries.

8.4 Dispatch

A MySQL Proxy wraps up the Qserv frontend so that queries can be submitted using any MySQL-compatible client or library. The frontend's generated queries are dispatched using two file-level transactions on Qserv's Xrootd cluster. The first transaction consists of opening a particular path for writing, writing the chunk query, and closing the file. The path contains a specified chunkId and has the format: xrootd://<manager_ip:port>/query2/CC, where CC is the chunkId. The second transaction consists of opening a path for reading, reading until EOF, and closing the file. The second path specifies the hash of the chunk query written in the original chunk query and has the format: xrootd://<worker_ip:port>/result/H, where H is the MD5 hash, represented via 32 hexadecimal digits in ASCII.

Chunk Query Representation

The format of a chunk query is given as a set of SQL query statements where the first line is a comment and indicates sub-chunk dependency.

```
-- SUBCHUNKS: <subChunkId0>[, <subChunkId1>[, ..]]
<SQL statement 1>;
[<SQL statement 2>];
...
```

The SUBCHUNKS line indicates the list of required sub-chunks for the query. The worker must generate the appropriate sub-chunk tables prior to executing the SQL statements, but is free to drop the tables afterwards. This enables the worker to cache sub-chunk tables, although the current implementation does not cache them.

Query Results Transfer. Results from a chunk query are transferred as SQL statements. The worker executes mysql_dump on the result table and the resulting byte stream is read byte-by-byte by the master, which executes the SQL statements to load results into its local database. After each result table is loaded, it is merged into a table which serves as the final result table for non-aggregating queries. When aggregation is needed, an aggregation query is executed on this table to produce the final result table.
Using `mysqldump` introduces overheads, but is the only user-level method provided by MySQL to transfer tables between database servers. We are considering implementing a more efficient method as development resources permit.

### 8.5 Aggregation

Qserv supports several SQL aggregation functions: AVG(), COUNT(), MAX(), MIN(), and SUM().

### 8.6 Indexing

By construction, Qserv's implementation of two-level spatial partitioning provides coarse spherical indexing so that spatially-restricted queries can execute involving only the relevant spatial fragments. However, access that is not spatially restricted involves the entire table by default. Qserv also implements indexing for one particular column, objectId. This is implemented by including a three-column table in the frontend's metadata database that maps objectId to chunkId and subChunkId. When a query predicated on objectId (the indexed column) is submitted, the frontend executes queries on this table to compute the containing set of chunks. Chunk tables on workers’ MySQL instances are also indexed by objectId so that indexed execution can be used on this containing set.

### 8.7 Cluster and Task Management

Qserv delegates management of cluster nodes to Xrootd. The Xrootd system manages cluster membership, node registration/de-registration, address lookup, replication, and communication. Its distributed filesystem API provides data-addressed communication channels to the rest of Qserv, hiding details like node count, the mapping of data to nodes, the existence of replicas, and node failure. The Qserv manager focuses on dispatching queries to endpoints and Qserv workers focus on receiving and executing queries on their local data.

### 8.8 Fault Tolerance

Qserv approaches fault tolerance in several ways. First, its design distributes responsibility so that components can be replicated and failures isolated. This technique is fundamental to Qserv’s incremental scalability and parallel performance. Second, components are connected with relatively narrow interfaces, minimizing component interdependence so that each part can operate (and fail) independently. Third, its software components contain logic for handling and recovering from errors.
Some components of Qserv are designed explicitly for fault-tolerance. The MySQL proxy is designed to balance load among several underlying MySQL servers and provide automatic fail-over when a server fails. The Xrootd distributed file system is designed to allow multiple managers and many redundant servers in the face of high request rate, high bandwidth, and unreliable hardware conditions. Qserv itself allows multiple masters to share the same redundant cluster of workers.

To illustrate Qserv's fault tolerance capabilities, we describe some failure conditions and Qserv's corresponding response mechanisms.

Consider the problem of a bug in Qserv's own code that, when triggered, crashes or hangs its process. If this were to happen on the worker, all query fragments belonging to that worker would be lost. If the process crashes, the queries in-flight on its mysqld will be cleaned up and resources freed. Monitoring software will detect the crash via heart-beat mechanism and restart the worker process. In the case of a freeze or hang, queries would remain in-flight on mysqld but unable to deliver results. A monitor could detect non-response and restart the server. In the worst case, the entire machine freezes and becomes unresponsive. The Xrootd system automatically handles this sort of failure and would silently direct new queries to different workers, provided the data are available elsewhere. The master would recognize that queries on the frozen machine are lost and retry. Should the failure happen on the master, the proxy could choose a different master.

A disk failure will be manifested similarly to a software fault and handled on a higher level as above. Qserv does not include logic to manage failure on localized regions of disks and would behave as if a software fault occurred. We are considering managing failure at a per-disk level, but would require research since application-level treatment of disk failure is relatively rare.

If a problem with the network occurs, the best outcome is that the fault is isolated to the now-unconnected machines. In Qserv, master-worker communication is orchestrated by Xrootd, which treats unresponsive servers as no longer able to accept requests. Thus network loss, server failure, or transient sluggishness/overload is treated similarly – work is directed elsewhere.

8.9 Current Status and Future Plans

As of now (August 2011) we have implemented a basic version of the system, capable of parsing selected queries, rewriting them into sub-queries and executing these sub-queries in parallel. We demonstrated running all query types (low, high, super-high) including aggregations, scalably on a 150-node cluster using 50 TB data set; we also demonstrated the system performs well enough to meet query response time meet the LSST requirements. A limited level of query concurrency (concurrent execution of 2 long-running queries and a stream of quick queries) was tested at scale.
This system is expected to be used for DC4 (DC3 data sets are too small to require Qserv technology)

Future work includes:
- troubleshooting and bug-fixing (concurrency problems, etc)
- examining and improving SQL syntax coverage
- a new more flexible and performant master/worker protocol
- a basic bulk table loader
- a unified cluster-state and table metadata data store
- design and feasibility evaluation for sub-query support
- user-level query management
- [--- all above needed before deploying for users ---]
- a basic shared scanning implementation
- table management
- basic support for user tables, including meta-operations
- demonstrating cross-match with external catalogs
- support for updates
- query fault recovery
- partition management
- support for HTM partitioning in Qserv
- authentication and authorization
- resource management
- early partition results
- performance improvements
- partition granularity varying per table.

We feel the first seven items have to be implemented before Qserv can be deployed for use by users other than core Qserv developers.

**Troubleshooting and bug fixing.** A subtle software bug related to a thread leak in the Qserv software prevented us from demonstrating concurrency at full-scale (beyond ~4 concurrent queries). Once the problem is fixed, we expect to re-run concurrency related queries as soon as we have access to a large testbed again (expected time scale: fall of 2011).

**Examining and improving SQL syntax coverage.** To-date, our focus centered around building and scaling the core components of the system, not on supporting the full range of SQL syntax. Before Qserv can be used by non-developers, it must be tested with a broad array of SQL syntax, and any syntax users might need must be handled:
1. in the short term: the commonly used syntax needs to be covered\(^\text{10}\) and all not-yet-supported syntax needs to be gracefully handled rather than allow system to crash or behave unexpectedly, and
2. in the long term all needed syntax needs to be fully supported.

**A new more flexible and performant master/worker protocol.** The current implementation uses unstructured text which must be parsed on both ends – such approach has been sufficient for a proof of concept prototype, however having a more flexible protocol will significantly simplify implementation of some essential features such as shared scans. This involves implementation of structured formats for master/worker communication (query dispatch, and results transfer).

**A basic bulk table loader.** Currently Qserv requires a developer to babysit and troubleshoot the loading process. The lack of any automated loader makes system setup a development task rather than an administrative one.

**A unified cluster-state and table metadata data store.** The current Qserv does not maintain any explicit system state, so it is currently difficult to generalize for different table partitioning, clustering configuration, and tables. The current system uses a mixture of configuration files, hardcoding, and developer intervention. A unified store would simplify the current mixture and take a step towards a system usable by non-developers.

**Design and feasibility evaluation for sub-query support.** Qserv does not support SQL sub-queries. Since there is evidence that such a capability might be useful to users, so we should formulate a few possible designs and understand how easy/difficult they would be to implement. Note that there are some alternative viable alternatives, such as splitting sub-queries into multiple queries, and/or using session variables. A naïve implementation that involves dumping all sub-query results to disk and then reading these results from disk, similarly to how multiple map/reduce stages are implemented, should be tractable to implement.

---

\(^{10}\) Some of the commonly used syntax needed in the short-term might be implemented through work-arounds – supporting full SQL grammar will be fully implemented in the long term (during the LSST Construction), but given the complexity of SQL grammar we anticipate that some features, such as sub-queries, would need to be worked-around in the short term, for example by splitting a query with sub-queries into multiple queries, or by blindly dumping sub-query result to disk and reading from disk, similarly to how multiple map/reduce stages are implemented.
A basic shared scanning implementation. Shared scanning can be implemented using scheduling algorithms on worker nodes. A simple algorithm selects queries for execution based on locality. The scheduler restricts queries in flight to those that access table partitions/chunk already cached/in-flight. The scheduler would control admission so that only one chunk per disk spindle is accessed at any time. Instead of choosing queries for execution, the scheduler chooses chunks. A history of chunks recently accessed is kept so that the scheduler avoids repeating chunks too soon. In this way, each worker can make scheduling choices for I/O-efficient processing without adhering to any centralized schedule or pre-determined processing sequence.

User-level query management. In terms of job control, Qserv does not provide means for inspecting or managing queries in-flight, and has no interface for halting queries except upon error detection. It is clear that users and administrators will need to check query status and possibly abort queries.

Table management. Qserv does not integrate support for inspecting or manipulating database and table metadata, except for a few primitive checks. Table management includes code for setting/changing table properties and keeping them consistent throughout the cluster.

Basic support for user tables, including meta-operations. The query system would be much more useful if users were allowed to maintain their own tables to store their own data or results from previous queries. They should be able to create, drop, and update their own tables within the system.

Demonstrating cross-match with external catalogs. One of the use cases involve cross matching with external catalogs. In case the catalogs to cross-match with is small, it will be treated as a small table and replicated as metadata tables will be. For cross-matching with larger catalogs, the catalog to cross-match with will need to be partitioned and distributed on the worker nodes.

Support for updates. Since the Object table is needed for alert production, and that table is too large to handle unpartitioned, Qserv will be needed in that context. Under those conditions, some support for updates would be needed, although it may be a bolt-on rather than a more integrated solution.

Query fault recovery. While Qserv can detect execution errors, it does not currently distinguish between availability, network, or query errors. Such a capability is tractable to implement and would allow Qserv to retry query fragments and gracefully handle node failures.
Partition management. Qserv needs some facility to manage partitioning—the parameters of existing partitioned data, distribution of partitions in the cluster, and the partitioning of ingested data. The current Qserv only works with data that is partitioned as a preparation step and loaded manually into a cluster that is configured with the same parameters. This is a manual process that is too fragile to be workable in a system with reasonable uptime.

Support for HTM partitioning in Qserv. HTM is an alternative to the rectangular box form of spatial partitioning currently implemented in Qserv. Since HTM allows for more advanced indexing and optimization, it may eventually replace the current partitioning algorithm.

Authentication and authorization. The current Qserv does not implement any form of security or privileges. All access is full access. A production database system should provide some facility of user or role-based access so that usage can be controlled and resources can be shared. This is in particular needed for Level-3 data products.

Resource management. A production system should have some way to manage/restrict resource usage and provide quality-of-service controls. This includes a monitoring facility that can track each node’s load and per-user-query resource usage.

Early partition results. When performing interactive exploration of an observational data set, users frequently issue large-scale queries that produce undesired results, even after testing such queries on small subsets of the data. We can ameliorate this behavior by providing the investigator with early partial results from the query, allowing the user to recognize that the returned values are incorrect and permitting the query to be aborted without wasting additional resources. There are two mechanisms we will implement in Qserv for providing early results. First, for queries that retrieve a filtered set of rows, matching rows can be returned as their query fragments complete, well before all fragments finish. Second, for queries that group, sort, or aggregate information and therefore perform a global operation after any per-partition processing, the global operation can be applied to increasingly large subsets of the per-partition results, returning an early partial result each time.

Performance improvements. Significant performance gain can be obtained by improving scheduler. These improvements pose interesting state of the art computing challenges; more details are available in Appendix D. In addition, some parts of Qserv are inefficient since they were implemented under constraints of development time rather than efficiency, or maintainability – rewriting them would result in further performance gains. Caching results for future queries is another example of performance optimization that can yield significant speed improvements.
Partitioning granularity varying per table. Since large tables in LSST vary significantly in row count and row size, it may be worthwhile to support partitioning with multiple granularities. For execution management it is useful to have partitions sized so that query fragments have similar execution cost. To achieve this, partitions may need different spatial sizes.

9. Large-scale Testing

9.1 Introduction

9.1.1 Ideal environment

Based on the detailed spreadsheet analysis, we expect the ultimate LSST production system will be composed of few hundred database servers [32.], so a realistic test should include a cluster of at least 100 nodes.

Total database size of a single data release will vary from ~400 TB (DR1) to ~10 PB (DR11). Realistic testing requires at least ~20-30 TB of storage (across all nodes).

Note that a lot of highly focused tests which are extremely useful to fine tune different aspects of the system can be done on a very small, 2-3 cluster, or even on a single machine. An example of that can be measuring the effect of table size on the performance of near-neighbor join: this type of join will be done per sub-partition, and sub-partitions will be small (few K rows), thus almost all tests involving a single sub-partition can be done on a single machine with very little disk storage.

A significant amount of testing should be done where the dataset size exceeds the system memory size by an order of magnitude. This testing is important to reveal system performance in the presence of disk performance characteristics.

It is essential to have at least two different types of catalogs: Object and Source. Of course this data needs to be correlated, that is, the objects should corresponds to the sources. Having these 2 tables will allow us to to measure speed of joins. It is not necessary to have other types of source-like tables (DiaSource, ForcedSource) – the tests done with Source should be a good approximation.

The most important characteristic of the test data is its spatial distribution. The data should reflect realistic densities: presence of very crowded or very sparse regions have influence on how data is partitioned and on performance of certain queries (e.g., speed of near neighbor inside one partition). Other than realistic spatial distribution, we need several fields to be valid (e.g., magnitudes) in order to try some queries with predicates.
These tests are not only used to prove our system is capable of meeting the requirements, but also as a mean to stress the system and uncover potential problems and bottlenecks. In practice, whoever runs these tests should well understand the internals of the scalable architecture system and turning MySQL.

9.1.2 Schedule of testing

- Selecting the base technology – before the end of Q2 2009
- Determining the architecture – before the end of Q3 2009
- Pre-alpha tests focused on parallel distributed query execution engine, tests at small scale (~20 nodes) – Q4 2009
- Most major features in (except shared scan, user tables), performance tests on mid-size cluster (~100 nodes) – before the end of 2010
- Scalability improvements and tests on a large cluster (150-250 nodes) – before the end of Q2 2011
- Performance improvements and tests on a large cluster (150-250 nodes) – before the end of 2011
- Ready for non-developer testing – end of Q2 2012
- Use tables, cross-match – end of 2012
- Shared scans – end of 2013
- Fault tolerance – end of Q2 2013
- Support for update – end of Q3 2013
- Large scale tests, performance tests on a large cluster – end of Q4 2013

For further details, see Docushare Document-11962.

9.1.3 Current status of tests

We have run several large scale tests, including a test with the “pre-alpha” version of our software written on top of MySQL, using the Tuson cluster at LLNL (160 nodes, each node: two Intel Xeon 2.4 GHz GPUs with 4 GB RAM and 1 local hard disk of 80 Gbs), and several 100-node tests run at SLAC [57]. These tests helped us uncover many bottlenecks and prompted rewriting parts of our software, as well as implementing several missing features in Xrootd.

The “best” test we run to-date included running Qserv in 40/100/150 node configurations, using 2 billion row Object and 32 billion row Source tables, total of 30 TB data set. We focus here on discussing that test.
9.2  150-node Scalability Test

9.2.1  Hardware
We configured a cluster of 150 nodes interconnected via gigabit Ethernet. Each node had 2 quad-core Intel Xeon X5355 processors with 16GB memory and one 500GB 7200RPM SATA disk. Tests were conducted with Qserv SVN r21589, MySQL 5.1.45 and Xrootd 3.0.2 with Qserv patches.

9.2.2  Data
We tested using a dataset synthesized by spatially replicating the dataset from a recent LSST data challenge (“PT1.1”). We used two tables: Object and Source\(^{11}\). These two tables are among the largest expected in LSST. Of these two, the Object table is expected to be the most frequently used. The Source table will have 50-200X the rows of the Object table, and its use is primarily confined to time series analyses that generally involve joins with the Object table.

The PT1.1 dataset covers a spherical patch with right-ascension between 358° and 5° and declination between -7° and 7°. This patch was treated as a spherical rectangle and replicated over the sky by transforming duplicate rows' RA and declination columns, taking care to maintain spatial distance and density by a non-linear transformation of right-ascension as a function of declination. This resulted in an Object table of 1.7 billion rows (2TB) and a Source table of 55 billion rows (30 TB)\(^{12}\). The Source table included only data between -54° and +54° in declination. The polar portions were clipped due to limited disk space on the test cluster. Partitioning was set for 85 stripes each with 12 sub-stripes giving a φ height of ~2.11° for stripes and 0.176° for sub-stripes. Each chunk thus spanned an area of approximately 4.5deg\(^2\), and each sub-chunk, 0.031deg\(^2\). This yielded 8,983 chunks. Overlap was set to 0.01667° (1 arc-minute).

9.2.3  Queries
The current Qserv development focus is on features for scalability. We have chosen a set of test queries that demonstrate performance for both cheap queries (interactive latency), and expensive queries (hour, day latency). Runs of low volume queries ranged from 15 to 20 queries, while runs of high volume queries and super high volume queries consisted of only a few or even one query due to their expense. All reported query times are according to the command-line MySQL client (MySQL).

\(^{11}\) The schema may be browsed online at http://lsst1.ncsa.uiuc.edu/schema/index.php?sVer=PT1_1

\(^{12}\) Source for the duplicator is available at http://dev.lsstcorp.org/trac/browser/DMS/qserv/master/trunk/examples
9.2.3.1 Low-volume 1 – object retrieval

SELECT * FROM Object WHERE objectId = <objId>

This query retrieves all information for a particular astronomical object. Queries of this type are expected to be very common. In testing, the objectId was randomized uniformly over the objects in the data set.

In Figure 1 we can see that performance of this query is roughly constant, taking about 4 seconds. Each run consisted of 20 queries. The slower performance of Runs 1 and 4, where each execution took 9 seconds, were probably the result of competing tasks in the cluster. We attribute the initial 8 second execution time in Run 5 and beyond to cold cache conditions (likely the objectId index) in the cluster.

9.2.3.2 Low-volume 2 – time series

SELECT taiMidPoint, fluxToAbMag(psfFlux), fluxToAbMag(psfFluxErr), ra, decl
FROM Source
WHERE objectId = <objId>

This query retrieves information from all detections of a particular astronomical object, effectively providing a time-series of measurements on a desired object. For testing, the objectId was randomized as for the Low Volume 1 query, which meant that null results were retrieved where the Source data was missing due to available space on the test cluster.
In Figure 2 we see that performance is roughly constant at about 4 seconds per query. Run 1 was done after Low Volume 1’s Run 1 and we discount its 9 second execution times similarly as anomalous.

9.2.3.3 Low-volume 3 – spatially-restricted filter

```
SELECT COUNT(*)
FROM Object
WHERE ra_PS BETWEEN 1 AND 2
  AND decl_PS BETWEEN 3 AND 4
  AND fluxToAbMag(zFlux_PS) BETWEEN 21 AND 21.5
  AND fluxToAbMag(gFlux_PS) - fluxToAbMag(rFlux_PS)
    BETWEEN 0.3 AND 0.4
  AND fluxToAbMag(iFlux_PS) - fluxToAbMag(zFlux_PS)
    BETWEEN 0.1 AND 0.12;
```

This query asks how many objects of a certain color exist within a square degree box in the sky. The spatial location was randomized uniformly within +/- 20° declination around the celestial equator. Limiting geospatial coverage is intended to limit performance variation due to varying object density that is a by-product of the spatial coverage of the original data set coupled with the simple data duplication technique we implemented. This query also exercises Qserv's rewriting of queries for simple aggregation.

In Figure 3 we see the same 4 second performance that was seen for the other low volume queries. Again, the ~9 second performance in Run 2 could not be reproduced so we discount it as resulting from competing processes on the cluster.

9.2.3.4 High volume 1 – count

```
SELECT COUNT(*) FROM Object
```
This simple query exercises Qserv's query execution engine and illustrates the built-in cost of querying over all partitions in the sky. In theory, execution could exploit Qserv's objectId index in order to produce an object count, but the current implementation does not rely on any centralized index. This COUNT(*) query was measured between 20-30 seconds, as shown in Figure 4. The slower performance during Run 1 can be attributed to interference of other processes (queries, maintenance) in the cluster.

9.2.3.5 High-volume 2 – full-sky filter

```
SELECT objectId, ra_PS, decl_PS, uFlux_PS, gFlux_PS, rFlux_PS, iFlux_PS, zFlux_PS, yFlux_PS
FROM Object
WHERE fluxToAbMag(iFlux_PS) - fluxToAbMag(zFlux_PS) > 4
```

This query retrieves all objects of a certain color beyond a threshold over the entire sky. It is a full table scan query over the Object table, and is an example of a simple query that would be batched into a shared-scan because of its I/O intensity. Figure 5 illustrates its stable performance over 150 nodes: 2.5 to 3 minutes per query. This may not be a fair measure of performance, since we have not controlled for caching behavior in MySQL and the operating system. The 7 minute time in Run 3 may be a more accurate measure of uncached execution time, and the shorter time a measure of overhead in a cached collection of the ~70k rows of results.
Using the on-disk data footprint (MySQL’s MyISAM .MYD, without indexes or metadata) of the Object table (1.824x10^{12} bytes), we can compute the aggregate effective table scanning bandwidth. Run 3’s 7 minute execution yields 4.0GB/s in aggregate, or 27MB/s per node, while the other runs yield approximately 11GB/s in aggregate, or 76MB/s per node. Since each node was configured to execute up to 4 queries in parallel, Run 3’s bandwidth is more realistic, given seek activity from competing queries and the disk manufacturer's reported theoretical transfer rate of 98MB/s.

9.2.3.6 High-volume 3 – density

\[
\text{SELECT COUNT(*) AS n, AVG(ra\_PS), AVG(decl\_PS), chunkId} \\
\text{FROM Object} \\
\text{GROUP BY chunkId}
\]

This query computes statistics for table fragments (which are roughly equal in spatial area), giving a rough estimate of object density over the sky. It illustrates more complex aggregation query support in Qserv. This query is of similar complexity to High Volume 2, but Figure 6 illustrates measured times significantly faster, which is probably due to reduced results transmission time. As mentioned for HV2, cache behavior was not controlled, but the 4 minute time in Run 3 may be close.

9.2.3.7 Super-high-volume 1 – near neighbor

\[
\text{SELECT COUNT(*)} \\
\text{FROM Object o1, Object o2} \\
\text{WHERE qserv\_areaspec\_box(-5,-5,5,-5)} \\
\text{AND qserv\_angSep(o1.ra\_PS, o1.decl\_PS, o2.ra\_PS, o2.decl\_PS) < 0.1}
\]

This query finds pairs of objects within a specified spherical distance which lie within a particular part of the sky. Over two randomly selected 100 deg^2 areas, the execution times were about 10 minutes (667.19 seconds and 660.25 seconds). The resultant row counts ranged between 3 to 5 billion. Since execution uses on-the-fly generated tables, the tables do not fit in memory, and Qserv does not yet implement caching, we expect caching effects to be negligible.
9.2.3.8 Super-high-volume 2 – sources not near objects

```
SELECT o.objectId, s.sourceId, s.ra, s.decl, o.ra_PS, o.decl_PS
FROM Object o, Source s
WHERE qserv_areaspec_box(224.1, -7.5, 237.1, 5.5)
    AND o.objectId = s.objectId
    AND qserv_angSep(s.ra, s.decl, o.ra_PS, o.decl_PS) > 0.0045
```

This is an expensive query – an $O(kn)$ join over 150 square degrees between a 2TB table and a 30TB table. Each objectId is unique in Object, but is shared by 41 rows (on average) in Source, so $k \approx 41$. We recorded times of a few hours (5:20:38.00, 2:06:56.33, and 2:41:03.45). The variance is presumed to be caused by varying spatial object density over the three random areas selected.

9.2.4 Scaling

We tested Qserv’s scalability by measuring its performance while varying the number of nodes in the cluster. To simulate different cluster sizes, the frontend was configured to only dispatch queries for partitions belonging to the desired set of cluster nodes. This varies the overall data size proportionally without changing the data size per node (200-300GB). We measured performance at 40, 100, and 150 nodes to demonstrate weak scaling.

**Scaling with small queries**

From Figure 7, 8, and 9, we see that execution time is unaffected by node count given that the data per node is constant. The spike in the 40-node configuration in Figure 8 is caused by 2 slow queries (23s and 57s); the other 28 executed in times ranging from 4.09 to 4.11 seconds.
Scaling with expensive queries

High Volume
If Qserv scaled perfectly linearly, the execution time should be constant when the data per node is constant. In Figure 10 the times for high volume queries show a slight increase. HV1 is a primarily a test of dispatch and result collection overhead and its time increases linearly with the number of chunks since the front-end has a fixed amount of work to do per chunk. Since we varied the set of chunks in order to vary the cluster size, the execution time of HV1 should thus vary linearly with cluster size. HV3 seems to have a similar trend since due to cache effects – its result was cached so execution became more dominated by overhead.

The High Volume 2 query approximately exhibits the flat behavior that would indicate perfect scalability. Caching effects may have clouded the results, but they did not dominate. If the query results were perfectly cached, we expect the overall execution time to be dominated by overhead as in HV1, and this is clearly not the case.

**Super High Volume**

The tests on expensive queries did not show perfect scalability, but nevertheless, the measurements did show some amount of parallelism. It is unclear why execution in the 100-node configuration was the slowest for both SHV1 and SHV2. Our time-limited access to the cluster did not allow us to repeat executions of these expensive queries and study their performance in better detail.

### 9.2.5 Concurrency

We were able to test Qserv with multiple queries in flight. We ran 4 “streams” of queries: two parallel invocations of HV2, one of LV1, and one of LV2. Each low volume stream paused for 1 second between queries.
Figure 12 illustrates concurrent performance. We see that the HV2 queries take about twice the time (5:53.75 and 5:53.71) as they would if running alone. This makes sense since each is a full table scan that is competing for resources and shared scanning has not been implemented. The first queries in the low volume streams execute in about 30 seconds, but each of their second queries seems to get “stuck” in queues. Later queries in the streams finish faster. Since the worker nodes maintain first-in-first-out queues for queries and do not implement any concept of query cost, long queries can easily hog the system. The slowness of low volume queries after the second queries may be curious at first glance, since they should be queued at the end on their assigned worker nodes and thus complete near the end of the HV2 queries. In that case, subsequent queries would land on workers with nearly empty queues and execute immediately. This slowness can be explained by query skew – short queries may land on workers that have or have not finished their work on the high volume queries.

9.2.6 Discussion

Latency

LSST's data access needs include supporting both small, frequent, interactive queries and longer, hour/day-scale queries. We designed Qserv to operate efficiently in both cases to avoid needing multiple systems, which would be costly in development, maintenance, and hardware. Indexing was implemented in order to reduce latency for cheap queries that only touch a small part of the data.

The current Qserv implementation incurs significant overhead in dispatching queries and collecting results. In early development we decided to minimize the intelligence on each worker, so the front-end master became responsible for preparing the SQL queries so that workers did not need to perform parsing or variable substitution. Results collection is somewhat heavyweight as well. MySQL does not provide a method to transfer tables between server instances, so tables are dumped to SQL statements using mysqldump and reloaded on the front-end. This method was chosen to speed prototyping, but its costs in speed, disk, network, and database transactions are strong motivations to explore a more efficient method.
Solid-state storage

Some of Qserv's design choices (e.g., shared scanning) are motivated by the need to work around poor seek performance characteristics of disks. Solid-state storage has now become a practical alternative to mechanical disk in many applications. While it may be useful for indexes, its current cost differential per unit capacity means that it is still impractical to store bulk data. In the case of flash storage, the most popular solid-state storage technology, shared scanning is still effective in optimizing performance since DRAM is much faster than flash storage and flash still has “seek” penalty characteristics (though it is much better than spinning disk).

Many core

We expect the performance to be I/O constrained, since the workload is data, not CPU performance limited. It is unlikely that many cores can be leveraged on a single node since they will be sized with only the number of disk spindles that saturate the north bridge, but shared scanning should increase CPU utilization efficiency.

Alternate partitioning

The rectangular fragmentation in right ascension and declination, while convenient to visualize physically for humans, is problematic due to severe distortion near the poles. We are exploring the use of a hierarchical scheme, such as the hierarchical triangular mesh [33.] for partitioning and spatial indexing. These schemes can produce partitions with less variation in area, and map spherical points to integer identifiers encoding the points' partitions at many subdivision levels. Interactive queries with very small spatial extent can then be rewritten to operate over a small set of fine partition IDs. If chunks are stored in partition ID order, this may allow I/O to occur at below sub-chunk granularity without incurring excessive seeks. Another bonus is that mature, well tested, and high-performance open source libraries exist for computing the partition IDs of points and mapping spherical regions to partition ID sets.

Distributed management

The Qserv system is implemented as a single master with many workers. This approach is reasonable and has performed adequately in testing, but the bottlenecks are clear. A Qserv instance at LSST's planned scale may have a million fragment queries in flight, and while we have plans to optimize the query management code path, managing millions from a single point is likely to be problematic. The test data set described in this paper is partitioned into about 9,000 chunks, which means that a launch of even the most trivial full-sky query launches about 9,000 chunk queries.
One way to distribute the management load is to launch multiple master instances. This is simple and requires no code changes other than some logic in the MySQL Proxy to load-balance between different Qserv masters. Another way is to implement tree-based query management. Instead of managing individual chunk queries, the master would dispatch groups of them to lower-level masters which would could either subdivide and dispatch subgroups or manage the individual chunk queries themselves.

10. References

1. *LSST Data Management Storage Sizing and I/O Model*, Docushare LDM-141 (spreadsheet) and LDM-139 (explanation)


4. XLDB website: [http://xldb.org](http://xldb.org)


6. *MapReduce: Simplified Data Processing on Large Clusters* – Google, Jeffrey Dean, Sanjay Ghemawat, OSDI’04


14. [http://dev.lsstcorp.org/trac/wiki/dbHiveExperiment](http://dev.lsstcorp.org/trac/wiki/dbHiveExperiment)


23. Lessons Learned from Managing a Petabyte, Jacek Becla, Daniel Wang, CIDR Conference, Asilomar, CA, USA, January 2005


29. Solid state disks tests, Docushare Document-11701

30. *DM Risk Register*, Docushare Document-7025

31. *Query Set for Performance Tests*, [http://dev.lsstcorp.org/trac/wiki/dbQueriesForPerfTest](http://dev.lsstcorp.org/trac/wiki/dbQueriesForPerfTest)

32. *LSST Data Management Infrastructure Costing*, Docushare LDM-144 (spreadsheet) and LDM-143 (explanation)

33. *The Hierarchical Triangular Mesh*, Peter Kunszt, Alexander Szalay, Aniruddha Thakar


47. [http://drizzle.org/](http://drizzle.org/)


52. Skype Plans for PostgreSQL to scale to 1 billion users, http://highscalability.com/skype-plans-postgresql-scale-1-billion-users
53. http://postgis.refractions.net/
55. Using Vertica as a Structured Data Repositories for Apache Hadoop, http://www.vertica.com/MapReduce
56. Gearman website: http://gearman.org/
57. 100-node scalability test run at SLAC, http://dev.lsstcorp.org/trac/wiki/dbQservTesting
11. Appendix A – Map/Reduce Solutions

11.1 Hadoop

Hadoop is a Lucene sub-project hosted by Apache. It is open source. It tries to re-create the Google MR technology [6.] to provide a framework in which parallel searches/projections/transformations (the Map phase) and aggregations/groupings/sorts/joins (the Reduce phase) using key-value pairs can be reliably performed over extremely large amounts of data. The framework is written in Java though the actual tasks executing the map and reduce phases can be written in any language as these are scheduled external jobs. The framework is currently supported for GNU/Linux platforms though there is on-going work for Windows support. It requires that ssh be uniformly available in order to provide daemon control.

Hadoop consists of over 550 Java classes implementing multiple components used in the framework:

- The Hadoop Distributed File System (HDFS), a custom POSIX-like file system that is geared for a write-once-read-many access model. HDFS is used to distribute blocks of a file, optionally replicated, across multiple nodes. HDFS is implemented with a single Namenode that maintains all of the meta-data (i.e., file paths, block maps, etc.) managed by one or more Datanodes (i.e., a data server running on each compute node). The Namenode is responsible for all meta-data operations (e.g., renames and deletes) as well as file allocations. It uses a rather complicated distribution algorithm to maximize the probability that all of the data is available should hardware failures occur. In general, HDFS always tries to satisfy read requests with data blocks that are closest to the reader. To that extent, HDFS also provides mechanisms, used by the framework, to co-locate jobs and data. The HDFS file space is layered on top of any existing native file system.
- A single JobTracker, essentially a job scheduler responsible for submitting and tracking map/reduce jobs across all of the nodes.
- A TaskTracker co-located with each HDFS DataNode daemon which is responsible for actually running a job on a node and reporting its status.
- DistributedCache to distribute program images as well as other required read-only files to all nodes that will run a map/reduce program.
- A client API consisting of JobConf, JobClient, Partitioner, OutputCollector, Reporter, InputFormat, OutputFormat among others that is used to submit and run map/reduce jobs and retrieve the output.
Hadoop is optimized for applications that perform a streaming search over large amounts of data. By splitting the search across multiple nodes, co-locating each with the relevant data, wherever possible, and executing all the sub-tasks in parallel, results can be obtained (relatively) quickly. However, such co-ordination comes with a price. Job setup is a rather lengthy process and the authors recommend that the map phase take at least a minute to execute to prevent job-setup from dominating the process. Since all of the output is scattered across many nodes, the map phase must also be careful to not produce so much output as to overwhelm the network during the reduce phase, though the framework provides controls for load balancing this operation and has a library of generally useful mappers and reducers to simplify the task. Even so, running ad hoc map/reduce jobs can be problematic. The latest workaround used by many Hadoop users involves running Hadoop services continuously (and jobs are attached to these services very fast). By default, joining tables in MR involves transferring data for all the joined tables into the reducer, and performing the join in the reduce stage, which can easily overwhelm the network. To avoid this, data must be partitioned, and data chunked joined together must be placed together (on the same node), in order to allow performing the join in the map stage.

Today's implementation of Hadoop requires full data scan even for simplest queries. To avoid this, indices are needed. Implementing indices has been planned by the community for several years, and according to the latest estimates they will be implemented in one or two years. In the meantime, those who need indices must implement and maintain them themselves, the index files can be stored e.g. as files in the Hadoop File System (HDFS).

One of the “features” of MR systems is lack of official catalog (schema); instead, knowledge about schema in part of the code. While this dramatically improves flexibility and speeds up prototyping, it makes it harder to manage such data store in the long term, in particular if multi-decade projects with large number of developers are involved.

Lack of some features that are at the core of every database system should not be a surprise – MR systems are simply built with different needs in mind, and even the Hadoop website officially states that Hadoop *is not a substitute for a database* [34.]. Nevertheless, many have attempted to compare Hadoop performance with databases. According to some publications and feedback from Hadoop users we talked to, Hadoop is about an order of magnitude more wasteful of hardware than a e.g. DB2 [35.].

Hadoop has a large community supporting it; e.g., over 300 people attended the first Hadoop summit (in 2008). It is used in production by many organizations, including Facebook, Yahoo!, and Amazon Facebook [36.] It is also commercially supported by Cloudera. Hadoop Summit 2011 was attended by more than 1,600 people from more than 400 companies.
We evaluated Hadoop in 2008. The evaluation included discussions with key developers, including Eric Baldeschwieler from Yahoo!, Jeff Hammerbacher from Facebook, and later Cloudera, discussions with users present at the 1st Hadoop Summit, and a meeting with the Cloudera team in September of 2010.

11.2 Hive
Hive [37.] is a data warehouse infrastructure developed by Facebook on top of Hadoop; it puts structures on the data, and defines SQL-like query language. It inherits Hadoop's deficiencies including high latency and expensive joins. Hive works on static data, it particular it can't deal with changing data, as row-level updates are not supported. Although it does support some database features, it is a “state where databases were ~1980: there are almost no optimizations” (based on Cloudera, meeting at SLAC Sept 2010). Typical solutions involve implementing missing pieces in user code, for example once can build their own indexes and interact directly with HDFS (and skip the Hadoop layer).

11.3 HBase
HBase [38.] is a column-oriented structured storage modeled after Google's Bigtable [7.], and built on top of the Hadoop HDFS. It is good at incremental updates and column key lookups, however, similarly to plain MR, it offers no mechanism to do joins – a typical solution used by most users is to denormalize data. HBase is becoming increasingly more popular at Facebook [39.]

11.4 Pig Latin
Pig Latin is a procedural data flow language for expressing data analysis programs. It provides many useful primitives including filter, foreach ... generate, group, join, cogroup, union, sort and distinct, which greatly simplify writing Map/Reduce programs or gluing multiple Map/Reduce programs together. It is targeted at large-scale summarization of datasets that typically require full table scans, not fast lookups of small numbers of rows. We have talked to the key developer of Pig Latin – Chris Olston.

11.5 Other Hadoop-related Systems
Other systems build for Hadoop include Zookeeper [40.] – a service for coordinating Hadoop's processes (ala Google's Chubby [8.]), and Simon – a cluster and application monitoring tool. Simon is similar to Ganglia, except it has more/better aggregation.
11.6 Dryad

Dryad [41., 42.] is a system developed by Microsoft Research for executing distributed computations. It supports a more general computation model than MR in that it can execute graphs of operations, using so called Directed Acyclic Graph (DAG). It is somewhat analogous to the MR model in that it can model MR itself, among others, more complex flows. The graphs are similar to the query plans in a relational database. The graph execution is optimized to take advantage of data locality if possible, with computation moving to the data. If non-local data is needed, it is transferred over the network.

Dryad currently works on flat files. It is similar to Hadoop in this way.

The core execution engine in Dryad has been used in production for several years but not heavily. There are several integration pieces we might want (loading data from databases instead of files, tracking replicas of data) that do not yet exist.

Beyond the execution engine, Dryad also incorporates a simple per-node task scheduler inherited from elsewhere in Microsoft. It runs prioritized jobs from a queue. Dryad places tasks on nodes based on the data available on the node and the state of the task queue on the node. A centralized scheduler might improve things, particularly when multiple simultaneous jobs are running; that is an area that is being investigated.

Dryad requires that the localization or partitioning of data be exposed to it. It uses a relatively generic interface to obtain this metadata from an underlying filesystem, enabling it to talk to either a proprietary GFS-like filesystem or local disks.

Dryad runs only on Windows .NET at present. Building the system outside of Microsoft is difficult because of dependencies on internal libraries; this situation is similar to the one with Google's GFS and Map/Reduce. The core execution engine could conceivably be implemented within Hadoop or another system, as its logic is not supposed to be terribly complicated. The performance-critical aspect of the system is the transfer of data between nodes, a task that Windows and Unix filesystems have not been optimized for and which Dryad therefore provides.

Dryad has been released as open source to academics/researchers in July 2009. This release however does not include any distributed filesystem for use with the system. Internally, Microsoft uses the Cosmos file system [43.], but it is not available in the academic release. Instead there are bindings for NTFS and SQL Server.
11.7 Dremel

Dremel is a scalable, interactive ad-hoc query system for analysis of read-only data, implemented as an internal project at Google [10]. Information about Dremel has been made available in July 2010. Dremel's architecture is in many ways similar to our baseline architecture (executing query in parallel on many nodes in shared nothing architecture, auto fail over, replicating hot spots). Having said that, we do not have access to the source code, even though Google is an LSST collaborator, and there is no corresponding open source alternative to date [44].

11.8 Tenzing

Tenzing is an SQL implementation on the MapReduce Framework [11] We managed to obtain access to pre-published paper from Google through our XLDB channels several months before planned publication at the VLDB 2011 conference.

Tenzing is a query engine built on top of MR for ad hoc analysis of Google data. It supports a mostly complete SQL implementation (with several extensions) combined with several key characteristics such as heterogeneity, high performance, scalability, reliability, metadata awareness, low latency support for columnar storage and structured data, and easy extensibility.

The Tenzing project underscores importance and need of database-like features in any large scale system.

11.9 "NoSQL"

The popular term NoSQL originally refered to systems that do not expose SQL interface to the user, and it recently evolved and refers to structured systems such as key-value stores or document stores. These systems tend to provide high availability at the cost of relaxed consistency (“eventual” consistency). Today’s key players include Cassandra [45] and MongoDB [45].

While a key/value store might come handy in several places in LSST, these systems do not address many key needs of the project. Still, a scalable distributed key-value store may be appropriate to integrate as an indexing solution within a larger solution.

12. Appendix B – Database Solutions

In alphabetical order.
12.1 Caché

InterSystems Caché is a shared-nothing object database system, released as an embedded engine since 1972. Internally it stores data as multi-dimensional arrays, and interestingly, supports overlaps. We are in discussions with the company—we have been discussing Caché with Stephen Angevine since early 2007, and met with Steven McGlothlin in June 2011. We also discussed Caché with William O'Mullane from the ESA's Gaia mission, an astronomical survey that selected Caché as their underlying database store in 2010 [24., 25.]). InterSystems offers free licensing for all development and research, for academic and non-profit research, plus support contracts with competitive pricing. However, their system does not support compression and stores data in strings, which may not be efficient for LSST catalog data.

A large fraction of the code is already available as open source for academia and non-profit organizations under the name “Globals” [58.].

12.2 DB2

IBM's DB2 “parallel edition” implements a shared-nothing architecture since mid-1990. Based on discussions with IBM representatives including Guy Lohman (e.g., at the meeting in June 2007) as well as based on various news, it appears that IBM's main focus is on supporting unstructured data (XML), not large scale analytics. All their majors projects announced in the last few years seem to confirm them, including Viper, Viper2 and Cobra (XML) and pureScale (OLTP).

12.3 Drizzle

Drizzle [47.] is a fork from the MySQL Database, the fork was done shortly after the announcement of the acquisition of MySQL by Oracle (April 2008). Drizzle is lighter than MySQL: most advanced features such as partitioning, triggers and many others have been removed (the code base was trimmed from over a million lines down to some 300K, it has also been well modularized). Drizzle's main focus is on the cloud market. It runs on a single server, and there are no plans to implement shared-nothing architecture. To achieve shared-nothing architecture, Drizzle has hooks for an opaque sharding key to be passed through client, proxy, server, and storage layers, but this feature is still under development, and might be limited to hash-based sharding.

Default engine is InnoDB. MyISAM engine is not part of Drizzle, it is likely MariaDB engine will become a replacement for MyISAM.

Drizzle’s first GA release occurred in March 2011.
We have discussed the details of Drizzle with key Drizzle architects and developers, including Brian Aker (the chief architect), and most developers and users present at the Drizzle developers meeting in April 2008.

### 12.4 Greenplum

Greenplum is a commercial parallelization extension of PostgreSQL. It utilizes a shared-nothing, MPP architecture. A single Greenplum database image consists of an array of individual databases which can run on different commodity machines. It uses a single Master as an entry point. Failover is possible through mirroring database segments. According to some, it works well with simple queries but has issues with more complex queries. Things to watch out for: distributed transaction manager, allegedly there are some issues with it.

Up until recently, Greenplum powered one of the largest (if not the largest) database setups: eBay was using it to manage 6.5 petabytes of data on a 96-node cluster [19.]. We are in close contact with the key eBay developers of this system, including Oliver Ratzesberger.

We are in contact with the Greenplum CTO: Luke Lonergan.

08/28/2008: Greenplum announced supporting MapReduce [48.]

Acquired by EMC in July 2010.

### 12.5 GridSQL

GridSQL is an open source project sponsored by EnterpriseDB. GridSQL is a thin layer written on top of postgres that implemented shared-nothing clustered database system targeted at data warehousing. This system initially looked very promising, so we evaluated it in more details, including installing it on our 3-node cluster and testing its capabilities. We determined that currently in GridSQL, the only way to distribute a table across multiple nodes is via hash partitioning. We can't simply hash partition individual objects, as this would totally destroy data locality, which is essential for spatial joins. A reasonable workaround is to hash partition entire declination zones (hash partition on zoneld), this will insure all objects for a particular zone end up on the same node. Further, we can “chunk” each zone into smaller pieces by using a regular postgres range partitioning (sub-tables) on each node.

The main unsolved problems are:

- near neighbor queries. Even though it is possible to slice a large table into pieces and distribute across multiple nodes, it is not possible to optimize a near neighbor query by taking advantage of data locality – GridSQL will still need to do $n^2$ correlations to complete the query. In practice a special layer on top of GridSQL is still needed to optimize near neighbor queries.
• shared scans.

Another issue is stability, and lack of good documentation.

Also since GridSQL is based on PostgreSQL, it inherits the postgres “cons”, such as the slow performance (comparing to MySQL) and having to reload all data every year.

The above reasons greatly reduce the attractiveness of GridSQL.

We have discussed in details the GridSQL architecture with their key developer, Mason Sharp, who confirmed the issues we identified are unlikely to be fixed/implemented any time soon.

**Gridsql Tests**

We installed GridSQL on a 3 node cluster at SLAC and run tests aimed to uncover potential bottlenecks, scalability issues and understand performance. For these tests we used simulated data generated by the software built for LSST by the UW team.

Note that GridSQL uses PostgreSQL underneath, so these tests included installing and testing PostgreSQL as well.

For these tests we used the USNO-B catalog. We run a set of representative queries, ranging from low volume queries (selecting a single row for a large catalog, a cone search), to high-volume queries (such as near-neighbor search).

Our timing tests showed acceptable overheads in performance compared to PostgreSQL standalone.

We examined all data partitioning options available in GridSQL. After reading documentation, interacting with GridSQL developers, we determined that currently in GridSQL, the only way to distribute a table across multiple nodes is via hash partitioning. We can't simply hash partition individual objects, as this would totally destroy data locality, which is essential for spatial joins. A reasonable workaround we found is to hash partition entire declination zones (hash partition on zoneId), this will insure all objects for a particular zone end up on the same node. Further, we can “chunk” each zone into smaller pieces by using a regular PostgreSQL range partitioning (sub-tables) on each node.

We were unable to find a clean solution for the near neighbor queries. Even though it is possible to slice a large table into pieces and distribute across multiple nodes, it is not possible to optimize a near neighbor query by taking advantage of data locality, so in practice GridSQL will still need to do \( n^2 \) correlations to complete the query. In practice a special layer on top of GridSQL is still needed to optimize near neighbor queries. So in practice, we are not gaining much (if anything) by introducing GridSQL into our architecture.

During the tests we uncovered various stability issues, and lack of good documentation.
In addition, GridSQL is based on PostgreSQL, so it inherits the PostgreSQL “cons”, such as the slow performance (comparing to MySQL) and having to reload all data every year, described separately.

### 12.6 InfiniDB

InfiniDB is an open source, columnar DBMS consisting of a MySQL front end and a columnar storage engine, built and supported by Calpont. Calpont introduced their system at the MySQL 2008 User Conference [50.], and more officially announced it in late Oct 2009 [51.]. It implements true MPP, shared nothing (or shared-all, depending how it is configured) DBMS. It allows data to be range-based horizontal partitioning, partitions can be distributed across many nodes (overlapping partitions are not supported though). It allows to run distributed scans, filter aggregations and hash joins, and offers both intra- and inter- server parallelism. During cross-server joins: no direct communication is needed between workers. Instead, they build 2 separate hash maps, distribute smaller one, or if too expensive to distribute they can put it on the “user” node.

A single-server version of InfiniDB software is available through free community edition. Multi-node, MPP version of InfiniDB is only available through commercial, non-free edition, and is closed source.

We are in contact with Jim Tommaney, CTO of the Calpont Corporation since April 2008. In late 2010 we run the most demanding query – the near neighbor tests using Calpont. Details of these tests are covered in Appendix C.

### 12.7 LucidDB

LucidDB is an open source columnar DBMS. Early startup (version 0.8 as of March 2009). They have no plans to do shared-nothing (at least there is no mention of it, and on their main page they mention “great performance using only a single off-the-shelf Linux or Windows server.”). Written mostly in java.

### 12.8 MySQL

MySQL utilizes a shared-memory architecture. It is attractive primarily because it is a well supported, open source database with a large company (now Oracle) behind it and a big community supporting it. (Note, however, that much of that community uses it for OLTP purposes that differ from LSST's needs.) MySQL's optimizer is below-average.

We have run many, many performance tests with MySQL. These are documented in trac in various places, many of them on these four pages:

- [http://dev.lsstcorp.org/trac/wiki/dbSpatialJoinPerf](http://dev.lsstcorp.org/trac/wiki/dbSpatialJoinPerf)
We are well plugged into the MySQL community, we attended all MySQL User Conferences in the past 5 years, and talked to many MySQL developers, including director of architecture (Brian Aker), the founders, and their optimizer gurus.

There are several notable open-source forks of MySQL: Percona (focusing on multi-core scaling), OurDelta (additional logging and backup on Percona), MariaDB (MySQL founder’s fork, focused on code quality + features), Drizzle (slimmed-down web-scale focus).

Spatial indexes / GIS. As of version 5.0.16, MySQL has included some support for spatial indexes and functions using the OpenGIS geometry model. We have not yet tested this portion of MySQL, and have preferred using geometry functionality from SciSQL, a MySQL plug-in written by a LSST contributor (Serge Monkewitz).

12.8.1 MySQL – Columnar Engines

**KickFire**

KickFire is a hardware appliance built for MySQL. It runs a proprietary database kernel (a columnar data store with aggressive compression) with operations performed on a custom dataflow SQL chip. An individual box can handle up to a few terabytes of data. There are several factors that reduce the attractiveness of this approach:

- it is a proprietary “black box”, which makes it hard to debug, plus it locks us into a particular technology
- it is an appliance, and custom hardware tends to get obsolete fairly rapidly
- it can handle low numbers of terabytes; another level is still needed (on top?) to handle petabytes
- there is no apparent way to extend it (not open source, all-in-one “black box”)

We have been in contact with the key people since April of 2007, when the team gave us a demo of their appliance under an NDA.

**InfoBright**

Infobright is a proprietary columnar engine for MySQL. Infobright Community Edition is open-source, but lacks many features, like parallelism and DML (INSERT, UPDATE, DELETE, etc). Infobright Enterprise Edition is closed-source, but supports concurrent queries and DML. Infobright’s solution emphasizes single-node performance without discussing distributed operation (except for data ingestion in the enterprise edition).
12.9 Netezza

Netezza Performance Server (NPS) is a proprietary, network attached, streaming data warehousing appliance that can run in a shared-nothing environment. It is built on PostgreSQL. The architecture of NPS consists of two tiers: a SMP host and hundreds of massively parallel blades, called Snippet Processing Units (SPU). Each SPU consists of a CPU, memory, disk drive and an FPGA chip that filters records as they stream off the disk. See http://www.netezza.com/products/index.cfm for more information.

According to some rumours, see eg http://www.dbms2.com/2009/09/03/teradata-and-netezza-are-doing-mapreduce-too/, Netezza is planning to support map/reduce.

Pros:
- It is a good, scalable solution
- It has good price/performance ratio.

Cons:
- it is an appliance, and custom hardware tends to get obsolete fairly rapidly
- high hardware cost
- proprietary

Purchased by IBM.

12.10 Oracle

Oracle provides scalability through Oracle Real Application Clusters (RAC). It implements a shared-storage architecture.

Cons: proprietary, expensive. It ties users into specialized (expensive) hardware (Oracle Clusterware) in the form of storage area networks to provide sufficient disk bandwidth to the cluster nodes; the cluster nodes themselves are often expensive shared-memory machines as well. It is very expensive to scale to very large data sets, partly due to the licensing model. Also, the software is very monolithic, it is therefore changing very, very slowly.

We have been approached several times by Oracle representatives, however given we believe Oracle is not a good fit for LSST, we decided not to invest our time in detailed investigation.

12.11 ParAccel

ParAccel Analytic Database is a proprietary RDBMS with a shared-nothing MPP architecture using columnar data storage. They are big on extensibility and are planning to support user-defined types, table functions, user-defined indexes, user-defined operators, user-defined compression algorithms, parallel stored procedures and more.
When we talked to ParAccel representatives (Rick Glick, in May 2008), the company was still in startup mode.

12.12 PostgreSQL

PostgreSQL is an open source RDBMS running in a shared-memory architecture.

PostgreSQL permits horizontal partitioning of tables. Some large-scale PostgreSQL-based applications use that feature to scale. It works well if cross-partition communication is not required.

The largest PostgreSQL setup we are aware of is AOL's 300 TB installation (as of late 2007). Skype is planning to use PostgreSQL to scale up to billions of users, by introducing a layer of proxy servers which will hash SQL requests to an appropriate PostgreSQL database server, but this is an OLTP usage that supports immense volumes of small queries [52].

PostgreSQL also offers good GIS support [53]. We are collaborating with the main authors of this extension.

One of the main weaknesses of PostgreSQL is a less-developed support system. The companies that provide support contracts are less well-established than Sun/MySQL. Unlike MySQL, but more like Hadoop, the community is self-organized with no single central organization representing the whole community, defining the roadmap and providing long term support. Instead, mailing lists and multiple contributors (people and organizations) manage the software development.

PostgreSQL is more amenable to modification than MySQL, which may be one reason why it has been used as the basis for many other products, including several mentioned below.

Based on the tests we run, PostgreSQL performance is 3.7x worse than MySQL. We realize the difference is partly due to very different characteristics of the engines used in these tests (fully ACID-compliant PostgreSQL vs non-transactional MyISAM), however the non-transactional solution is perfectly fine, and actually preferred for our immutable data sets.

We are in touch with few most active PostgreSQL developers, including the authors of Q3C mentioned above, and Josh Berkus.

Tests

We have run various performance tests with PostgreSQL to compare its performance with MySQL. These tests are described in details in the “Baseline architecture related” section below. Based on these tests we determined PostgreSQL is significantly (3.7x) slower than MySQL for most common LSST queries.
We have also tried various partitioning schemes available in PostgreSQL. In that respect, we determined PostgreSQL is much more advanced than MySQL.

Also, during these tests we uncovered that PostgreSQL requires dump/reload of all tables for each major data release (once per year), see [http://www.postgresql.org/support/versioning](http://www.postgresql.org/support/versioning). The PostgreSQL community believes this is unlikely to change in the near future. This is probably the main show-stopper preventing us from adapting PostgreSQL.

### 12.13 SciDB

SciDB is a new open source system inspired by the needs of LSST\(^{13}\) and built for scientific analytics. SciDB implements a shared nothing, columnar MPP array database, user defined functions, overlapping partitions, and many other features important for LSST. SciDB Release 11.06, the first production release, was published on June 15, 2011. We are in the process of testing this release.

### 12.14 SQLServer

Microsoft's SQLServer's architecture is shared-memory. The largest SQLServer based setup we are aware of is the SDSS database (6 TB), and the Pan-STARRS database.

In 2008 Microsoft bought DATAllegro and began an effort, codenamed “Project Madison,” to integrate it into SQLServer. Madison relies on shared nothing computing. Control servers are connected to compute nodes via dual Infiniband links, and compute servers are connected to a large SAN via dual Fiber Channel links. Fault tolerance relies on (expensive) hardware redundancy. For example, servers tend to have dual power supplies. However, servers are unable to recover from storage node failures, thought a different replica may be used. The only way to distribute data across nodes is by hashing; the system relies on replicating dimension tables. [the above is based on the talk we attended: [http://wiki.esi.ac.uk/w/files/5/5c/Dyke-Details_of_Project_Madison-1.pdf](http://wiki.esi.ac.uk/w/files/5/5c/Dyke-Details_of_Project_Madison-1.pdf)]

Cons: It is proprietary, relies on expensive hardware (appliance), and it ties users to the Microsoft OS.

**About DATAllegro.** DATAllegro was a company specializing in data warehousing server appliances that are pre-configured with a version of the Ingres database optimized to handle relatively large data sets (allegedly up to hundreds of terabytes). The optimizations reduce search space during joins by forcing hash joins. The appliances rely on high speed interconnect(Infiniband).

\(^{13}\) See [http://scidb.org/about/history.php](http://scidb.org/about/history.php)
12.15 Sybase IQ
Sybase IQ [54.] is a commercial columnar database product by Sybase Corp. Sybase IQ utilizes a “shared-everything” approach that designed to provide graceful load-balancing. We heard opinions that most of the good talent has left the company; thus it is unlikely it will be a major database player.

Cons: proprietary.

12.16 Teradata
Teradata implements a shared-nothing architecture. The two largest customers include eBay and WalMart. Ebay is managing multi petabyte Teradata-based database.

The main disadvantage of Teradata is very high cost.

We are in close contact with Steve Brobst, acting as Teradata CTO, and key database developers at eBay.

12.17 Vertica
The Vertica Analytics Platform is a commercial product based on the open source C-store column-oriented database, and now owned by HP. It utilizes a shared-nothing architecture. Its implementation is quite innovative, but involves significant complexity underneath.

It is built for star/snowflake schemas. It currently can not join multiple fact tables; e.g. self-joins are not supported though this will be fixed in future releases. Star joins in the MPP environment are made possible by replicating dimension tables and partitioning the fact table.

In 2009, a Vertica Hadoop connector was implemented. This allows Hadoop developers to push down map operators to Vertica database, stream Reduce operations into Vertica [55.], and move data between the two environments.

Cons:
- lack of support of self-joins
- proprietary.
12.18 Others

In addition to map/reduce and RDBMS systems, we also evaluation several other software packages which could be used as part of our custom software written on top of MySQL. The components needed include SQL parser, cluster management and task management.

12.18.1 Cluster and task and management

Two primary candidates to use as cluster and task management we identified so far are gearman and Xrootd. Cluster management involves keeping track of available nodes, allowing nodes to be added/removed dynamically. Task management involves executing tasks on the nodes.

Detailed requirements what we need are captured at:
http://dev.lsstcorp.org/trac/wiki/dbDistributedFrameworkRequirements

Gearman

Gearman is a distributed job execution system, available as open source. It provides task management functions, e.g., cluster management is left out to be handled in application code.

During a meeting setup in June 2009 with Eric Day, the key developer working on integration of Drizzle with Gearman, who also wrote the C++ version of Gearman, we discussed details of Gearman architecture and its applicability for LSST.

Gearman manages workers as resources that provide RPC execution capabilities. It is designed to provide scalable access to many resources that can provide similar functionality (e.g., compress an image, retrieve a file, perform some expensive computation). While we could imagine a scheme to use Gearman’s dispatch system, its design did not match LSST’s needs well. One problem was its store-and-forward approach to arguments and results, which would mean that the query service would need to implement its own side transmission channel or potentially flood the Gearman coordinator with bulky results.

13. Appendix C: Tests with InfiniDB

In late 2010 we collaborated with the Calpont team on testing their InfiniDB product. Testing involved executing the most complex queries such as near neighbor on 1 billion row USNOB catalog. The tests were run by Jim Tommaney, the final results are pasted below.

Thank you for the chance to evaluate InfiniDB against the stellar data set and the near neighbor problem. Towards that end I installed our 2.0 version on a Dell 610 server with 16GB memory, 8 physical cores (16 Hyper-Threaded Intel virtual cores), and a 4 disk raid 0 data mount point with 7200 RPM disk drives.

As you know, the N-squared search space becomes problematic at scale, so part of the
solution involved a specialized query and the addition of 4 additional columns as shown below. These new columns defined two overlapping grids on top of the search space such that any given object existed in 2 grids. Note that these are abstract grids represented by additional columns and the table is a single table with our standard vertical + horizontal partitioning that happens with the basic create table statement. So, this virtual 'gridding' doesn't change other characteristics of the table or prevent other such extensions.

Column additions:
alter table object add column ra_r2 decimal(5,2);
alter table object add column decl_r2 decimal(4,2);
alter table object add column ra_d2 decimal(5,2);
alter table object add column decl_d2 decimal(4,2);

Example update statements:
update object set ra_r2 = round(ra,2) where ra < 10;
update object set decl_r2 = round(decl,2) where ra < 10;
update object set ra_d2 = truncate(ra,2) where ra < 10;
update object set decl_d2 = truncate(decl,2) where ra < 10;

The query itself consist of 4 parts, the sum of which is the count of near neighbors.

1. Search within Grid D defined by ra_d2, decl_d2.
2. Search within Grid R defined by ra_r2, decl_r2, adding predicates to only include matches that span two D Grids.
   
   \[ \text{and} \left( ( o1.ra_d2 \neq o2.ra_d2 ) \text{ or } ( o1.decl_d2 \neq o2.decl_d2 ) \right) \]

3. There is the additional condition where a given pair of neighbors span both Grid D and Grid R. For this subset of the data, the neighboring objects share Grid R coordinates for RA, and Grid D coordinates for Decl.
   
   \[
   \text{FROM object o1 join object o2 on}
   \text{(o1.ra_r2 = o2.ra_r2 and o1.decl_d2 = o2.decl_d2)}
   \]

4. The last case covers the same basic condition as 3, but includes a join that covers neighboring objects that share Grid D coordinates for RA, and Grid R coordinates for Decl.
   
   \[
   \text{FROM object o1 join object o2 on}
   \text{(o1.ra_d2 = o2.ra_d2 and o1.decl_r2 = o2.decl_r2)}
   \]
Anyway, the results appear very promising, and indicate that it may satisfy arbitrarily large search spaces. I executed the query against the full range of declination, and searched with the range of RA between 0 and ra_limit. I then scaled ra_limit between 0.01 through 20 and charted the results below, trending search space vs. rows processed per second. The baseline numbers you provided appear to avg. about 1000 rows/second, and capped out at about 80k search space. With InfiniDB, the search rate is relatively flat after a ramp-up of a couple seconds, running at about ~800 K rows processed per second through a search space of about 32 M x 32 M objects. At 32M objects x 32M objects the query consumed about 6GB for the hash structures, however extending the query logic above would allow for running something like 33 of these queries serially to search through a 1B x 1B space. Running the 4 sections serially would reduce the memory requirements if desired.

There are a number of variations on the near neighbor problem that provide a filter on one of the object tables, i.e. search for white dwarf that I would characterize as M x N problems where M << N. To profile those queries I selected an arbitrary filter (o1.bMag > 24.9) that restricted 1 of the 2 sides of the join to at most ~330 K objects. I then executed 12 queries with ra between 0 and ra_limit, varying ra_limit from 30 to 360. Each query was executed 3 times sequentially following a flush of the data buffer cache, and the average of the three
values charted.

With a bounded M, the processing rate went up significantly, approaching 4.5 M rows per second when the second and third executions of a query were satisfied from cache, and running at nearly 3 M rows per second for larger queries that did not fit in the data buffer cache (which was configured at 8 GB). These queries only used about 6% of memory for temporary space and could be run against an arbitrarily large N as desired.

![InfiniDB M x N selective query](image)

There are more details regarding the load rate, options on other grid sizes, limitation of this style grid analysis for larger definitions of 'near', etc. that can be shared and reviewed as desired, and I am more than happy to profile additional queries as desired. For example, I can take a look at getting an exact time for finding all of the near-neighbors within a 1B x 1B search space if that is interesting (should be something like 23-25 minutes), it is just a matter of tweaking the query restrictions to allow proper handling of objects on each side of these larger query boundary.

There are definitely some significant differences between InfiniDB and MySQL in terms of best practices for a number of items. For example, our fastest load capability is via cpimport rather than load data infile. The near neighbors problem appears to be one example of many where we handle large data analysis significantly better than MySQL,
although there are plenty of examples where MySQL shines relative to InfinDB (individual row insertion, individual record access via index, etc). Any external application that relies on individual row lookups with an expected latency in the microseconds will run significantly slower with InfiniDB.

```sql
select sum(cnt) from (  
    SELECT count(*) cnt  
    FROM object o1 join object o2 using(ra_d2, decl_d2)  
    WHERE ABS(o1.ra - o2.ra) < 0.00083 / o2.cosRadDecl  
        AND ABS(o1.decl - o2.decl) < 0.00083  
        AND o1.objectid <> o2.objectid  
        AND o1.ra >= 0 AND o1.ra < @ra_limit  
        AND o1.bMag > 24.9  
        AND o2.ra >= 0 AND o2.ra < @ra_limit  
        AND (o1.ra_d2 <> o2.ra_d2)  
    union all  
    SELECT count(*) cnt  
    FROM object o1 join object o2 using(ra_r2, decl_r2)  
    WHERE ABS(o1.ra - o2.ra) < 0.00083 / o2.cosRadDecl  
        AND ABS(o1.decl - o2.decl) < 0.00083  
        AND o1.objectid <> o2.objectid  
        AND o1.ra >= 0 AND o1.ra < @ra_limit  
        AND o1.bMag > 24.9  
        AND o2.ra >= 0 AND o2.ra < @ra_limit  
        AND (o1.ra_r2 <> o2.ra_r2)  
    union all  
    select count(*) cnt  
    FROM object o1 join object o2 on (o1.ra_r2 = o2.ra_r2  
        and o1.decl_d2 = o2.decl_d2  
        and abs(o1.ra - o1.ra_r2) * o1.cosRadDecl < 0.00083  
        and abs(o2.ra - o2.ra_r2) * o2.cosRadDecl < 0.00083  
        and abs(o1.decl - (o1.decl_d2 + 0.005)) < 0.00083  
        and abs(o2.decl - (o2.decl_d2 + 0.005)) < 0.00083  
        and o1.objectid <> o2.objectid  
        AND o1.ra >= 0 AND o1.ra < @ra_limit  
        AND o1.bMag > 24.9  
        AND o2.ra >= 0 AND o2.ra < @ra_limit  
    union all  
    select count(*) cnt  
    FROM object o1 join object o2 on (o1.ra_d2 = o2.ra_d2  
        and o1.decl_r2 = o2.decl_r2  
        and abs(o1.ra - o1.ra_d2) * o1.cosRadDecl < 0.00083  
        and abs(o2.ra - o2.ra_d2) * o2.cosRadDecl < 0.00083  
        and abs(o1.decl - (o1.decl_r2 + 0.005)) < 0.00083  
        and abs(o2.decl - (o2.decl_r2 + 0.005)) < 0.00083  
        and o1.objectid <> o2.objectid  
        AND o1.ra >= 0 AND o1.ra < @ra_limit  
        AND o1.bMag > 24.9  
        AND o2.ra >= 0 AND o2.ra < @ra_limit  
```
and abs(o1.ra - (o1.ra_d2 + 0.005)) * o1.cosRadDecl < 0.00083
and abs(o2.ra - (o2.ra_d2 + 0.005)) * o2.cosRadDecl < 0.00083
and abs(o1.decl - o1.decl_r2 ) < 0.00083
and o1.objectid <> o2.objectid
and o1.ra >= 0 and o1.ra < @ra_limit and o1.bMag > 24.9
and o2.ra >= 0 and o2.ra < @ra_limit
) a;

mysql> set @ra_limit:= 0.01;
Query OK, 0 rows affected (0.00 sec)

mysql> \. near_neighbors.sql
+----------+------------------------+--------------------------+
<table>
<thead>
<tr>
<th>count(*)</th>
<th>count(distinct(ra_d2))</th>
<th>count(distinct(decl_d2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>16652</td>
<td>1</td>
<td>8643</td>
</tr>
</tbody>
</table>
+----------+------------------------+--------------------------+
1 row in set (0.07 sec)

+----------+
| sum(cnt) |
+----------+
|     2834 |
+----------+
1 row in set (0.60 sec)

+---------------------------------------+
<table>
<thead>
<tr>
<th>idb()</th>
</tr>
</thead>
</table>
+---------------------------------------+
| Query Stats: MaxMemPct-0; ApproxPhyI/O-0; CacheI/O-0; BlocksTouched-0; PartitionBlocksEliminated-0 |  
+---------------------------------------+
1 row in set (0.00 sec)

mysql> set @ra_limit:= 0.1;
Query OK, 0 rows affected (0.00 sec)

mysql> \. near_neighbors.sql

83
+----------+------------------------+--------------------------+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
+----------+------------------------+--------------------------+
|   167632 |                     10 |                    17048 |
+----------+------------------------+--------------------------+
1 row in set (0.08 sec)

+----------+
| sum(cnt)  |
+----------+
|    31757 |
+----------+
1 row in set (0.95 sec)

mysql> set @ra_limit:= 0.5;
Query OK, 0 rows affected (0.00 sec)

mysql> \near_neighbors.sql
+----------+------------------------+--------------------------+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
+----------+------------------------+--------------------------+
|   833849 |                     50 |                    17812 |
+----------+------------------------+--------------------------+
1 row in set (0.16 sec)

+----------+
| sum(cnt)  |
+----------+
|   155065 |
+----------+
1 row in set (2.07 sec)
mysql> set @ra_limit:= 1;
Query OK, 0 rows affected (0.00 sec)

mysql> \. near_neighbors.sql
+----------+------------------------+--------------------------+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
+----------+------------------------+--------------------------+
| 1638966 |                    100 |                    17891 |
+----------+------------------------+--------------------------+
1 row in set (0.21 sec)

+----------+
| sum(cnt) |
+----------+
|     290972 |
+----------+
1 row in set (2.22 sec)

mysql> set @ra_limit:= 2;
Query OK, 0 rows affected (0.00 sec)
mysql> \. near_neighbors.sql
+-----------------+----------------+-----------------+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
|-----------+---------------------------+---------------------------|
| 3153437 | 200 | 17947 |
+-----------------+----------------+-----------------+
1 row in set (0.25 sec)

+----------+
| sum(cnt) |
+----------+
| 516670 |
+----------+
1 row in set (3.79 sec)

| idb() |
|
+-------+
| Query Stats: MaxMemPct-8; ApproxPhyI/O-0; CacheI/O-0; BlocksTouched-0; PartitionBlocksEliminated-0 |
+-------+
1 row in set (0.00 sec)

mysql> set @ra_limit:= 3;
Query OK, 0 rows affected (0.00 sec)

mysql> \. near_neighbors.sql
+-----------------+----------------+-----------------+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
|-----------+---------------------------+---------------------------|
| 4675828 | 300 | 17961 |
+-----------------+----------------+-----------------+
1 row in set (0.28 sec)

+----------+
| sum(cnt) |
+----------+
| 734540 |
+----------+
1 row in set (6.51 sec)
+
| idb() |
| +

---
| Query Stats: MaxMemPct-9; ApproxPhyI/O-0; CacheI/O-0; BlocksTouched-0; PartitionBlocksEliminated-0 |
+
---
1 row in set (0.01 sec)

mysql> set @ra_limit:= 4;
Query OK, 0 rows affected (0.00 sec)

mysql> \ . near_neighbors.sql
+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
| +
| 6356011 | 400 | 17967 |
+
1 row in set (0.40 sec)
+
| sum(cnt) |
| +
| 1037556 |
+
1 row in set (7.61 sec)
+
---
| idb() |
| +

---
| Query Stats: MaxMemPct-13; ApproxPhyI/O-0; CacheI/O-0; BlocksTouched-0; PartitionBlocksEliminated-0 |
+
---
1 row in set (0.00 sec)
```
mysql> set @ra_limit:= 5;
Query OK, 0 rows affected (0.00 sec)

mysql> . near_neighbors.sql
+----------+------------------------+--------------------------+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
+----------+------------------------+--------------------------+
|  7989071 |                      500 |                    17975 |
+----------+------------------------+--------------------------+
1 row in set (0.46 sec)

+----------+
| sum(cnt)  |
+----------+
| 1326087  |
+----------+
1 row in set (9.38 sec)

mysql> . near_neighbors.sql
+----------+------------------------+--------------------------+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
+----------+------------------------+--------------------------+
| 15896387 |                     1000 |                    17986 |
+----------+------------------------+--------------------------+
1 row in set (0.92 sec)

mysql> set @ra_limit:= 10;
Query OK, 0 rows affected (0.00 sec)
```

mysql> set @ra_limit:= 15;
Query OK, 0 rows affected (0.00 sec)

mysql> \.

  near_neighbors.sql

+----------+------------------------+--------------------------+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
+----------+------------------------+--------------------------+
| 24045205 |                   1500 |                    17993 |
+----------+------------------------+--------------------------+
1 row in set (1.34 sec)

+---------+
| sum(cnt) |
+---------+
| 3949531 |
+---------+
1 row in set (32.46 sec)

89
1 row in set (0.01 sec)

mysql> set @ra_limit:= 20;
Query OK, 0 rows affected (0.00 sec)

mysql> \.
ear_neighbors.sql

+----------+------------------------+--------------------------+
| count(*) | count(distinct(ra_d2)) | count(distinct(decl_d2)) |
+----------+------------------------+--------------------------+
| 31841849 |                   2000 |                    17996 |
+----------+------------------------+--------------------------+
1 row in set (1.74 sec)

+----------+
| sum(cnt) |
+----------+
| 5247760 |
+----------+
1 row in set (40.48 sec)

select sum(cnt) from (  
SELECT count(*) cnt  
FROM object o1 join object o2 using(ra_d2, decl_d2)  
WHERE  ABS(o1.ra - o2.ra) < 0.00083 / o2.cosRadDecl AND ABS(o1.decl - o2.decl) < 0.00083  and o1.objectid < o2.objectid  
and o1.ra >= 0 and o1.ra < @ra_limit  
and o2.ra >= 0 and o2.ra < @ra_limit  
union all  
90
SELECT  count(*) cnt
FROM object o1 join object o2 using(ra_r2, decl_r2)
WHERE  ABS(o1.ra - o2.ra) < 0.00083 / o2.cosRadDecl
and ABS(o1.decl - o2.decl) < 0.00083
and o1.objectid < o2.objectid
and o1.ra >= 0 and o1.ra < @ra_limit
and o2.ra >= 0 and o2.ra < @ra_limit
and (( o1.ra_d2 <> o2.ra_d2 ) or (o1.decl_d2 <> o2.decl_d2))
union all
select count(*) cnt
    FROM object o1 join object o2 on (o1.ra_r2 = o2.ra_r2  and
o1.decl_d2 = o2.decl_d2 )
    WHERE  ABS(o1.ra - o2.ra) < 0.00083 / o2.cosRadDecl AND
    ABS(o1.decl - o2.decl) < 0.00083
and o1.ra_d2 <> o2.ra_d2
and o1.decl_d2 <> o2.decl_d2
and abs(o1.ra - o1.ra_r2) * o1.cosRadDecl < 0.00083
and abs(o2.ra - o2.ra_r2) * o2.cosRadDecl < 0.00083
and abs(o1.decl - (o1.decl_d2 + 0.005)) < 0.00083
and abs(o2.decl - (o2.decl_d2 + 0.005)) < 0.00083
and o1.objectid < o2.objectid
and o1.ra >= 0 and o1.ra < @ra_limit
and o2.ra >= 0 and o2.ra < @ra_limit
union all
select count(*) cnt
    FROM object o1 join object o2 on (o1.ra_d2 = o2.ra_d2  and
o1.decl_r2 = o2.decl_r2 )
    WHERE  ABS(o1.ra - o2.ra) < 0.00083 / o2.cosRadDecl AND
    ABS(o1.decl - o2.decl) < 0.00083
and o1.ra_r2 <> o2.ra_r2
and o1.decl_r2 <> o2.decl_r2
and abs(o1.ra - (o1.ra_d2 + 0.005)) * o1.cosRadDecl < 0.00083
and abs(o2.ra - (o2.ra_d2 + 0.005)) * o2.cosRadDecl < 0.00083
and abs(o1.decl - o1.decl_r2 ) < 0.00083
and abs(o2.decl - o2.decl_r2 ) < 0.00083
and o1.objectid < o2.objectid
and o1.ra >= 0 and o1.ra < @ra_limit
and o2.ra >= 0 and o2.ra < @ra_limit
) a;
14. Appendix D Qserv-related Research Topics

**Locality-efficient group scheduling:** Qserv breaks up a query into a number of independent sub-queries with unique data dependencies. When multiple high-level queries are in flight, it must schedule many more sub-queries, trying to minimize the latency of each top-level query while maximizing the total system hardware utilization and I/O efficiency. We propose researching this problem of locality-efficient group scheduling and implementing the solution in the context of Qserv. We will characterize its performance and describe the trade-offs that affect system performance.

**Windowed-mapping of distributed request queues on k-resources:** Because Qserv queries can be split into thousands (or millions) of independent sub-queries, the resulting task of assigning sub-queries to nodes may be computationally expensive. We propose development of a windowed-mapping algorithm that ensures $O(k)$ assignment given a sufficient window size, as well as heuristics for finding the window size for a given system configuration. Successful development of the algorithm and corresponding implementation will help Qserv scale beyond thousands of nodes with near constant latency.

**Fault-tolerant sloppy-state task management:** In large clusters, the task manager is often bottlenecked by the frequency and sheer number of state updates from its numerous execution nodes. We propose development of a scheduling algorithm that shifts part of the management to the requesting clients and keeping only sloppy system state on the centralized scheduler. We speculate that stale state information is usually sufficient for the scheduler, and that client-reported corrections in the remaining cases is low enough that keeping sloppy state is a large net benefit. We believe this management model will minimize the management messaging between the scheduler, clients, and workers and enable extreme scaling in Qserv. Our successful implementation should be useful for many other systems for scheduling tasks on large numbers of execution nodes.

15. Appendix E: People/Communities We Talked To

Solution providers of considered products:

- Map/Reduce – key developers from Google
- Hadoop – key developers from Yahoo!
- Hadoop - founders and key developers behind Cloudera, a company supporting enterprise edition of Hadoop
- Hive – key developers from Facebook. (RDBMS system written on top of Hadoop)
• Dryad – key developers from Microsoft (Dryad is Microsoft's version of map/reduce), including Michael Isard
• Gearman – key developers (gearman is a system which allows to run MySQL in a distributed fashion)
• representatives from all major database vendors, including Teradata, Oracle, IBM/DB2, Greenplum, Postgres, MySQL, MonetDB
• representatives from promising startups including HadoopDB, ParAcell, EnterpriseDB, Calpont, Kickfire
• Intersystem's Cache—Stephen Angevine, Steven McGlothlin

User communities
• developers from eBay. They use an RDBMS in production, but they did run extensive tests of Hadoop, as well as other systems including Vertica and Greenplum
• developers from the Amazon data warehouse team
• Nokia,
• AOL
• science users from HEP (LHC), astronomy (SDSS, Gaia, 2MASS, DES, Pan-STARRS, LOFAR), geoscience, biology

Leading database researchers
• M Stonebraker
• D DeWitt
• S Zdonik
• D Maier
• M Kersten