LARGE SYNOPTIC SURVEY TELESCOPE

Large Synoptic Survey Telescope (LSST)

Data Management Database Design

Jacek Becla, Daniel Wang, Serge Monkewitz, K-T Lim, John Gates, Andy Salnikov, Andrew Hanushevsky, Douglas Smith, Bill Chickering, Michael Kelsey, and Fritz Mueller

LDM-135

Latest Revision: 2017-07-06

Abstract

This document discusses the LSST database system architecture.

Change Record

Version	Date	Description	Owner name
1.0	2009-06-15	Initial version.	Jacek Becla
2.0	2011-07-12	Most sections rewritten, added scalability test section	Jacek Becla
2.1	2011-08-12	Refreshed future-plans and schedule of testing sections, added section about fault tolerance.	Jacek Becla, Daniel Wang
3.0	2013-08-02	Synchronized with latest changes to the requirements (LSE-163). Rewrote most of the "Implementation" chapter. Documented new tests, refreshed all other chapters.	Jacek Becla, Daniel Wang, Serge Monkewitz, Kian-Tat Lim, Douglas Smith, Bill Chickering
3.1	2013-10-10	Refreshed numbers based on latest LDM-141. Updated shared scans (implementation) and 300-node test sections, added section about shared scans demonstration	Jacek Becla, Daniel Wang
3.2	2013-10-10	TCT approved	R Allsman
	2016-07-18	Update with async query, shared scan, secondary index, xrootd, metadata service information.	John Gates, Andy Salnikov, Andrew Hanushevsky, Michael Kelsey, Fritz Mueller
	2017-07-05	Move historical investigations to separate documents.	T. Jenness

Contents

Data Management Database Design

1 Executive Summary

The LSST baseline database architecture for its massive user query access system 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 horizontally 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. Selected tables are also partitioned vertically to maximize performance of most common analysis. The system uses a few critical 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.

The LSST baseline database architecture for its real time Alert Production relies on horizontal time-based partitioning. To guarantee reproducibility, no-overwrite-update techniques combined with maintaining validity time for appropriate rows are employed. Two database replicas are maintained to isolate live production catalogs from user queries; the replicas are synchronized in real time using native database replication.

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. 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[1] [9] as an open source, elastic, distributed, fault-tolerant messaging system.

We developed a prototype of 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 variety of clusters ranging from 20-node-100TB cluster to 300-node-30TB cluster, tables as large as 50 billion rows and concurrency exceeding 100,000 in-flight chunk-queries. Required data rates for all types of queries (interactive, full sky scans, joins, correlations) have been achieved. Basic fault tolerance recovery mechanisms and basic version of shared scans were demonstrated. Future work includes adding support for user tables, 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. The largest barrier preventing us from using an off-the-shelf system is spherical geometry and spherical partitioning support.

To increase the chances such a system will become reality in the next few years, in 2008 we initiated the SciDB array-based scientific database project. Due to lack of many traditional RDBMS-features in SciDB and still nascent fault tolerance, we believe it is easier to build the LSST database system using MySQL+XRootD than it would be to build it based on SciDB. In addition we closely collaborate with the MonetDB open source columnar database team – a successful demonstration of Qserv based on MonetDB instead of MySQL was done in 2012. 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[2] conference series we lead, and beyond.

[1] `http://xrootd.org`
[2] `https://xldb.org`

2 Introduction

This document discusses the LSST database system architecture. Section 3 discusses the baseline architecture. Section 4 explains the LSST database-related requirements. Section 5 discusses design trade-offs and decision process, including small scale tests we run. Section 6 covers risk analysis. Section 7 discusses the prototype design (Qserv), including design, current status of the software and future plans. Section 8 and Section 9 describe large scale Qserv tests and demonstrations. For some additional background, DMTN-046 covers in-depth analysis of off-the-shelf potential solutions (Map/Reduce and RDBMS) as of 2013.

3 Baseline Architecture

This section describes the most important aspects of the LSST baseline database architecture. The choice of the architecture is driven by the project requirements (see reqs) as well as cost, availability and maturity of the off-the-shelf solutions currently available on the market (see DMTN-046), and design trade-offs (see Section 5). The architecture is periodically revisited: we continuously monitor all relevant technologies, and accordingly fine-tune the baseline architecture.

In summary, the LSST baseline architecture for Alert Production is an off-the-shelf RDBMS system which uses replication for fault tolerance and which takes advantage of horizontal (time-based) partitioning. The baseline architecture for user access to Data Releases 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,[3] 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

Alert Production involves detection and measurement of difference-image-analysis sources (DiaSources). New DiaSources are spatially matched against the most recent versions of existing DiaObjects, which contain summary properties for variable and transient objects (and

[3] A chunk's overlap is implicitly contained within the overlaps of its edge sub-chunks.

false positives). Unmatched DiaSources are used to create new DiaObjects. If a DiaObject has an associated DiaSource that is no more than a month old, then a forced measurement (DiaForcedSource) is taken at the position of that object, whether a corresponding DiaSource was detected in the exposure or not.

The output of Alert Production consists mainly of three large catalogs – DiaObject, DiaSource, and DiaForcedSource - as well as several smaller tables that capture information about e.g. exposures, visits and provenance.

These catalogs will be modified live every night. After Data Release Production has been run based on the first six months of data and each year's data thereafter, the live Level 1 catalogs will be archived to tape and replaced by the catalogs produced by DRP. The archived catalogs will remain available for bulk download, but not for queries.

Note that existing DiaObjects are never overwritten. Instead, new versions of the AP-produced and DRP-produced DiaObjects are inserted, allowing users to retrieve (for example) the properties of DiaObjects as known to the pipeline when alerts were issued against them. To enable historical queries, each DiaObject row is tagged with a validity start and end time. The start time of a new DiaObject version is set to the observation time of the DiaSource or DiaForcedSource that led to its creation, and the end time is set to infinity. If a prior version exists, then its validity end time is updated (in place) to equal the start time of the new version. As a result, the most recent versions of DiaObjects can always be retrieved with:

```
SELECT * FROM DiaObject WHERE validityEnd = infinity
```

Versions as of some time *t* are retrievable via:

```
SELECT * FROM DiaObject WHERE validityStart <= t AND t < validityEnd
```

Note that a DiaSource can also be reassociated to a solar-system object during day time processing. This will result in a new DiaObject version unless the DiaObject no longer has any associated DiaSources. In that case, the validity end time of the existing version is set to the time at which the reassociation occurred.

Once a DiaSource is associated with a solar system object, it is never associated back to a DiaObject. Therefore, rather than also versioning DiaSources, columns for the IDs of both the associated DiaObject and solar system object, as well as a reassociation time, are included. Reassociation will set the solar system object ID and reassociation time, so that DiaSources for DiaObject 123 at time *t* can be obtained using:

```
SELECT *
FROM DiaSource
WHERE diaObjectId = 123
AND midPointTai <= t
AND (ssObjectId is NULL OR ssObjectReassocTime > t)
```

DiaForcedSources are never reassociated or updated in any way.

From the database point of view then, the alert production pipeline will perform the following database operations 189 times (once per LSST CCD) per visit (every 39 seconds):

1. Issue a point-in-region query against the DiaObject catalog, returning the most recent versions of the objects falling inside the CCD.
2. Use the IDs of these diaObjects to retrieve all associated diaSources and diaForcedSources.
3. Insert new diaSources and diaForcedSources.
4. Update validity end times of diaObjects that will be superseded.
5. Insert new versions of diaObjects.

All spatial joins will be performed on in-memory data by pipeline code, rather than in the database. While Alert Production does also involve a spatial join against the Level 2 (DRP-produced) Object catalog, this does not require any database interaction: Level 2 Objects are never modified, so the Object columns required for spatial matching will be dumped to compact binary files once per Data Release. These files will be laid out in a way that allows for very fast region queries, allowing the database to be bypassed entirely.

The DiaSource and DiaForcedSource tables will be split into two tables, one for historical data and one containing records inserted during the current night. The current-night tables will be small and rely on a transactional engine like InnoDB, allowing for speedy recovery from failures. The historical-data tables will use the faster non-transactional MyISAM or Aria storage engine, and will also take advantage of partitioning. The Data Release catalogs used to seed the live catalogs will be stored in a single initial partition, sorted spatially (using the Hierarchical Triangular Mesh trixel IDs for their positions). This means that the diaSources and

diaForcedSources for the diaObjects in a CCD will be located close together on disk, minimizing seeks. Every month of new data will be stored in a fresh partition, again sorted spatially. Such partitions will grow to contain just a few billion rows over the course of a month, even for the largest catalog. At the end of each night, the contents of the current-night table are sorted and appended to the partition for the current-month, then emptied. Each month, the entire current-month partition is sorted spatially (during the day), and a partition for the next month is created.

For DiaObject, the same approach is used. However, DiaObject validity end-time updates can occur in any partition, and are not confined to the current-night table. We therefore expect to use a transactional storage engine like InnoDB for all partitions. Because InnoDB clusters tables using the primary key, we will likely declare it to consist of a leading HTM ID column, followed by disambiguating columns (diaObjectId, validityStart). The validity end time column will not be part of any index.

No user queries will be allowed on the live production catalogs. We expect to maintain a separate replica just for user queries, synchronized in real time using one-way master-slave native database replication. The catalogs for user queries will be structured identically to the live catalogs, and views will be used to hide the splits (using a "`UNION ALL`").

For additional safety, we might choose to replicate the small current-night tables, all DiaObject partitions, and the remaining (small) changing tables to another hot stand-by replica. In case of disastrous master failure that cannot be fixed rapidly, the slave serving user queries will be used as a temporary replacement, and user queries will be disallowed until the problem is resolved.

Based on the science requirements, only short-running, relatively simple user queries will be needed on the Level 1 catalogs. The most complex queries, such as large-area near neighbor queries, will not be needed. Instead, user queries will consist mainly of small-area cone searches, light curve lookups, and historical versions of the same. Since the catalogs are sorted spatially, we expect to be able to quickly answer spatial queries using indexed HTM ID columns and the scisql UDFs, an approach that has worked well in data-challenges to date. Furthermore, note that the positions of diaSources/diaForcedSources associated with the same diaObject will be very close together, so that sorting to obtain good spatial locality also ends up placing sources belonging to the same light curve close together. In other words, the data organization used to provide fast pipeline query response is also advantageous for user queries.

3.2 Data Release Production

Data Release Production will involve the generation of significantly larger catalogs than Alert Production. However, these are produced over the course of several months, pipelines will not write directly to the database, and there are no pipeline queries with very low-latency execution time requirements to be satisfied. While we do expect several pipeline-related full table scans over the course of a Data Release Production, we will need to satisfy many user queries involving such scans on a daily basis. User query access is therefore the primary driver of our scalable database architecture, which is described in detail below. For a description of the data loading process, please see qserve-data-loading.

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

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.

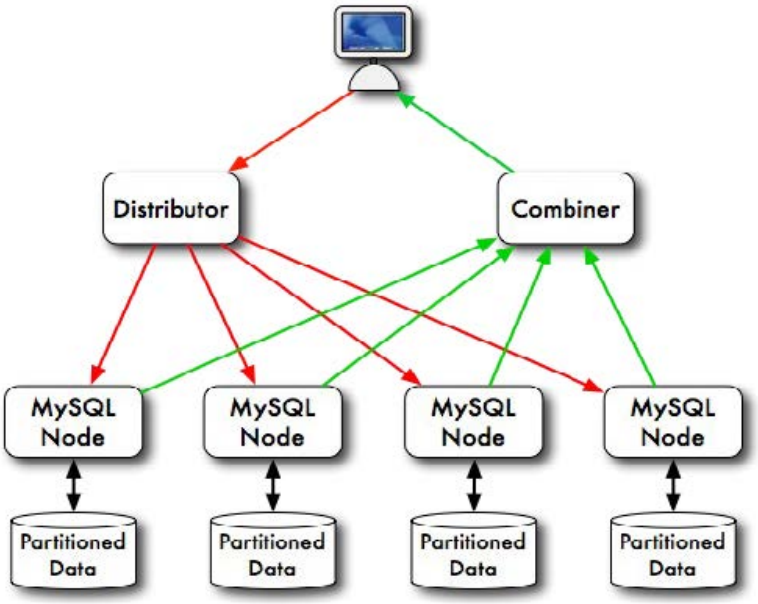

FIGURE 1: Shared-nothing database architecture.

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 advantages 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. For an analytical query system, it makes sense to make as few assumptions as possible about what will be important to our users, and to try and provide reasonable performance for as broad a query load as possible, i.e. focus on scan throughput rather than optimizing indexes. A further benefit to this approach is that many different queries are likely to be able to share scan

IO, boosting system throughput, whereas caching index lookup results is likely to provide far fewer opportunities for sharing as the query count scales (for the amounts of cache we can afford).

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 also lowers the cost of data compression by amortizing the CPU cost among the sharing queries, tilting the tradeoff of increased CPU cost versus reduced I/O cost heavily in favor of compression.

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:

- one full table scan of Object table for the latest data release only,
- one synchronized full table scan of Object, Source and ForcedSource tables every 12 hours for the latest data release only,
- one synchronized full table scan of Object and Object_Extra every 8 hours for the latest and previous data releases.

Appropriate Level 3 user tables will be scanned as part of each shared scan as needed to answer any in-flight user queries.

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 [LDM-141] 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 LDM-141.

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, DiaForcedSource) 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.

SSObject catalog will be unpartitioned, because there is no obvious fixed position that we could choose to use for partitioning. The associated diaSources (which will be intermixed with diaSources associated with static diaSources) will be partitioned, according their position. For that reason the SSObject-to-DiaSource join queries will require index searches on all chunks, unlike DiaObject-to-DiaSource queries. Since SSObject is small (low millions), this should not be an issue.

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.

3.3.6.1 Sharded data and sharded queries All catalogs that require spatial partitioning (Object, Source, ForcedSource, DiaSource, DiaForcedSource) 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.

3.3.6.2 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.

3.3.6.3 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 pre-computed 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.

3.3.6.4 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.

3.3.6.5 Data immutability It is important to note that user query access operates on read-only data. Not having to deal with updates simplifies the architecture and allows us to add extra optimizations not possible otherwise. The Level 1 data which is updated is small enough and will not require the scalable architecture – we plan to handle all Level 1 data set with out-of-the box MySQL as described in alert-production.

3.3.7 Long-running queries

Many of the typical user queries may need significant time to complete, at the scale of hours. To avoid re-submission of those long-running queries in case of various failures (networking or hardware issues) the system will support asynchronous query execution mode. In this mode users will submit queries using special options or syntax and the system will dispatch a query and immediately return to user some identifier of the submitted query without blocking user session. This query identifier will be used by user to retirieve query processing status, query result after query completes, or a partial query result while query is still executing.

The system should be able to estimate the time which user query will need to complete and refuse to run long queries in a regular blocking mode.

3.3.8 Technology choice

As explained in DMTN-046, 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. 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.

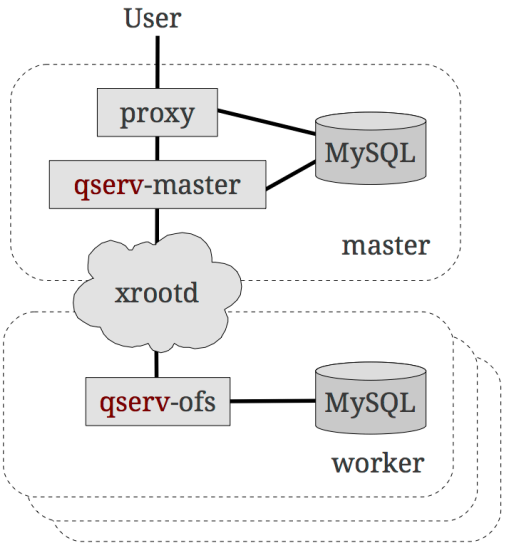

FIGURE 2: Component connections in Qserv.

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 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, uncompressed, data only) are captured in the table below <tab-expected-catalog-size>. For further storage, disk and network bandwidth and I/O analyses, see LDM-141.

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 updates the DiaObject, SSObject, DiaSource, DiaForcedSource 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 row updates/ingested is modest: ~40K new rows and updates occur every ~39 sec [6].
- 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 Level 1 live catalog will need to be updated with minimal delay. This catalog should not be taken off-line for extended periods of time.

The database system must allow for occasional schema changes for the Level 1 data, and occasional changes that do not alter query results[4] for the Level 2 data after the data has been released. Schemas for different data releases are allowed to be very different.

4.3 Query Access Related Requirements

The Science Data Archive Data Release query load is defined primarily in terms of access to the large catalogs in the archive: Object, Source, and ForcedSource. Queries to image meta-

[4] Example of non-altering changes including adding/removing/resorting indexes, adding a new column with derived information, changing type of a column without loosing information, (eg., `FLOAT` to `DOUBLE` would be always allowed, DOUBLE to `FLOAT` would only be allowed if all values can be expressed using `FLOAT` without loosing any information)

data, for example, though numerous, are expected to be fast and can easily be handled by replicating the relatively small metadata tables.

The large catalog query load is specified as follows:

1. 100 simultaneous queries for rows corresponding to single Objects or small spatial regions (on the order of at most 10s of arcminutes), with each query having an average response time of 10 seconds. This leads to a throughput of 10 "low-volume" queries per second. This number is approximately five times the peak "professional astronomer" query rate to the SDSS SkyServer. Each low-volume query is expected to return 0.5 GB of data or less. These queries are further subdivided as follows:

   A. Single object fetches: 5%.
   B. Few objects fetched by objectId: 60%.
   C. Small area by spatial index: 25%.
   D. Small area by scan: 10%.

   Furthermore, 70% of queries are expected to be of Objects only, with 20% retrieving Sources for Objects and 10% retrieving ForcedSources.

2. 50 simultaneous analytical queries involving full table scans of one or more large tables, with a target throughput of 20 queries per hour. Each "high-volume" query is expected to return up to 6 GB of data. We further subdivide these as follows:

   A. Throughput of 16 queries per hour with an average latency of 1 hour on the most frequently accessed columns in the Object table. These provide fast turnaround and high throughput for the most common types of queries.
   B. Throughput of 1 query per hour with average latency of 12 hours for joins of the Source table with the most frequently accessed columns in the Object table. These provide a reasonable turnaround time and good throughput for time series queries.
   C. Throughput of 1 query per hour with average latency of 12 hours for joins of the ForcedSource table with the most frequently accessed columns in the Object table. These provide a reasonable turnaround time and good throughput for detailed time series queries.
   D. Throughput of 1 query per hour with average latency of 8 hours for scans of the full Object table or joins of it with up to three additional tables other than Source and ForcedSource. These provide "adhoc" access for complex queries.

E. Throughput of 1 query per hour with average latency of 8 hours for scans of the full Object table in the previous Data Release or joins of it with up to three additional tables. These provide "ad hoc" access for older data.

We also include in the requirements up to 20 simultaneous queries for the Level 1 Database and 5 simultaneous queries for the Engineering and Facilities Database, both completing in an average of 10 seconds.

Reproducibility. Queries executed on any Level 1 and Level 2 data products must be reproducible.

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 ~100 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 ~8-12 hours. ~50 simultaneous high-volume queries are expected to be running at any given time.

Cross-matching with external/user data. 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.

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. Many end users will want to express queries directly in SQL, most of basic SQL92 will be required. 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

4.4.1 Implications

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 in both spatial and temporal dimensions 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.

4.4.2 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 [4]. 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.

External data sets and user data, including results from past queries may have to be distributed alongside distributed production table to provide adequate join performance.

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

5 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 section covers the test we run to determine the optimal architecture of MySQL-based system.

5.1 Standalone Tests

5.1.1 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), pre-calculating 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 in [2]. We found that PostgreSQL was 3.7x slower for spatial joins over a range of row counts, and reducing the row-count per partition to less than 5k rows was crucial in achieving lowering compute intensity, but that predicate selectivity could compensate for a 2-4x greater row count.

5.1.2 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/db/BuildSubPart`. These tests showed that it was fastest to build the on-the-fly sub-partitions using SQL in the engine, rather than performing the task externally and loading the sub-partitions back into the engine.

5.1.3 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/db/SubPartOverhead`. The on-the-fly overhead was measured to be 18% for `select *` queries, but 3600% if only one column (the skinniest selection) was needed.

5.1.4 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.

5.1.5 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` This revealed that single 1-billion row table usage is adequate in loading and indexing, but query performance was only acceptable when the query predicates selectivity using an index was a small absolute number of rows (1% selectivity is too loose). Thus a large fraction of index-scans were unacceptably slow and the table join speed was also slow.

5.1.6 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 `https://dev.lsstcorp.org/trac/wiki/MyIsamCompression`. We found that table data compressed only 50%, but since indexes were not compressed, there was only about 20% space savings. Table scans are significantly slower due to CPU expense, but short, indexed queries were only impacted 40-50%.

5.1.7 Full table scan performance

To determine performance of full table scan, we measured:

1. raw disk speed with `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 `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 *73% of the raw disk speed* (using MySQL MyISAM).

5.1.8 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:

  SELECT * **from** Object **where** objectId=<xx>
  SELECT * **from** Source **where** objectId =<xx>

- 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.:

```
SELECT * FROM Object
WHERE ra BETWEEN :raMin AND :raMax
AND decl BETWEEN :declMin AND :declMax
AND zMag BETWEEN :zMin AND :zMax
```

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 [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.

5.1.9 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 Document-11701. We found that concurrent query execution is dominated by software inefficiencies when solid-

state devices (SSDs) with fast random I/O are substituted for slow disks. Because the cost per byte is higher for SSDs, spinning disks are cheaper for bulk storage, as long as access is mostly sequential (which can be facilitated with shared scanning). However, because the cost per random I/O is much lower for SSDs than for spinning disks, using SSDs for serving indexes, exposure metadata, perhaps even the entire Object catalog, as well as perhaps for temporary storage is advised. This is true for the price/performance points of today's SSDs. Yet even with high IOPS performance from SSDs, table-scan based selection is often faster than index-based selection: a table-scan is faster than an index scan when >9% of rows are selected (cutoff is >1% for spinning disk). The commonly used 30% cutoff does not apply for large tables for present storage technology.

5.2 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 detail in Section 8.

5.2.1 DC1: data ingest

We ran 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 Document-1386. We found that as long as indexes are disabled during loading, ingest speed is typically CPU bound due to data conversion from ASCII to binary format. We also found that ingest into InnoDB is usually ~3x slower than into MyISAM, independently of table size.

5.2.2 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: `https://dev.lsstcorp.org/trac/wiki/db/DC2/PartitioningTests` and the pages linked from there. We determined that we need to maintain a narrow subset of the data, and fetch it from disk to memory right before the time-critical association in order to minimize database-related delays.

5.2.3 DC3: catalog construction

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

5.2.4 Winter-2013 Data Challenge: querying database for forced photometry

Prior to running Winter-2013 Data Challenge, we tested performance of MySQL to determine whether the database will be able to keep up with forced photometry production which runs in parallel. We determined that a single MySQL server is able to easily handle 100-200 simultaneous requests in well under a second. As a result we chose to rely on MySQL to supply input data for forced photometry production. Running the production showed it was the right decision, e.g., the database performance did not cause any problems. The test is documented at `https://dev.lsstcorp.org/trac/wiki/db/tests/ForcedPhoto`.

5.2.5 Winter-2013 Data Challenge: partitioning 2.6 TB table for Qserv

The official Winter-2013 production database, as all past data challenged did not rely on Qserv, instead, plain MySQL was used instead. However, as an exercise we partitioned and loaded this data set into Qserv. This data set relies on table views, so extending the administrative tools and adding support for views inside Qserv was necessary. In the process, administrative tools were improved to flexibly use arbitrary number of batch machines for partitioning and loading the data. Further, we added support for partitioning RefMatch* tables; RefMatch objects and sources have to be partitioned in a unique way to ensure they join properly with the corresponding Object and Source tables.

5.2.6 Winter-2013 Data Challenge: multi-billion-row table

The early Winter 2013 production resulted in 2.6 TB database; the largest table, ForcedSource, had nearly 4 billion rows.[5] Dealing with multi-billion row table is non-trivial are requires special handling and optimizations. Some operations, such as building an index tend to take a long time (tens of hours), and a single ill-tuned variable can result in 10x (or worse) performance degradation. Producing the final data set in several batches was in particular challenging, as we had to rebuild indexes after inserting data from each batch. Key lessons learned have been documented at `https://dev.lsstcorp.org/trac/wiki/mysqlLargeTables`. Issues we uncovered with MySQL (myisamchk) had been reported to the MySQL developers, and were fixed immediately fixed.

In addition, some of the more complex queries, in particular these with spatial constraints had to be optimized.[6] The query optimizations have been documented at `https://dev.lsstcorp.org/trac/wiki/db/MySQL/Optimizations`.

6 Risk Analysis

6.1 Potential Key Risks

Insufficient **database performance and scalability** is one of the major risks [Document-7025].

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

[5] It is worth noting that in real production we do not anticipate to manage billion+ rows in a *single physical table* - the Qserv system that we are developing will split every large table into smaller, manageable pieces.

[6] Some of these optimizations will not be required when we use Qserv, as Qserv will apply them internally.

largest single LSST data set have been successfully demonstrated in production today. For example, eBay manages a 10+ petabyte production database [14] 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).[7] 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. We believe the largest barrier preventing us from using an off-the-shelf DBMS such as InfiniDB is spherical geometry and spherical partitioning support.

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[8] (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).

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 anoma-

[7] The numbers, both for eBay and LSST are for compressed data sets.
[8] Now abandoned: `https://en.wikipedia.org/wiki/Drizzle_%28database_server%29`
[9] With the exception of Drizzle, which introduced major changes to the architecture.

lies 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 patters. 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.

The complete list of all database-related risks maintained in the LSST risk registry:

- DM-014: Database performance insufficient for planned load
- DM-015: Unexpected database access patterns from science users
- DM-016: Unexpected database access patterns from DM productions
- DM-032: LSST DM hardware architecture becomes antiquated
- DM-060: Dependencies on external software packages
- DM-061: Provenance capture inadequate
- DM-065: LSST DM software architecture incompatible with de-facto community standards
- DM-070: Archive sizing inadequate
- DM-074: LSST DM software architecture becomes antiquated
- DM-075: New SRD requirements require new DM functionality

6.2 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 [4] 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 [3]. 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.

7 Implementation of the Query Service (Qserv) Prototype

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 Section 3.

7.1 Components

7.1.1 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.

7.1.2 XRootD

The 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 over 10 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 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 as an XRootD client to workers by writing to partition-addressed XRootD paths and reads results from hash-addressed XRootD paths.

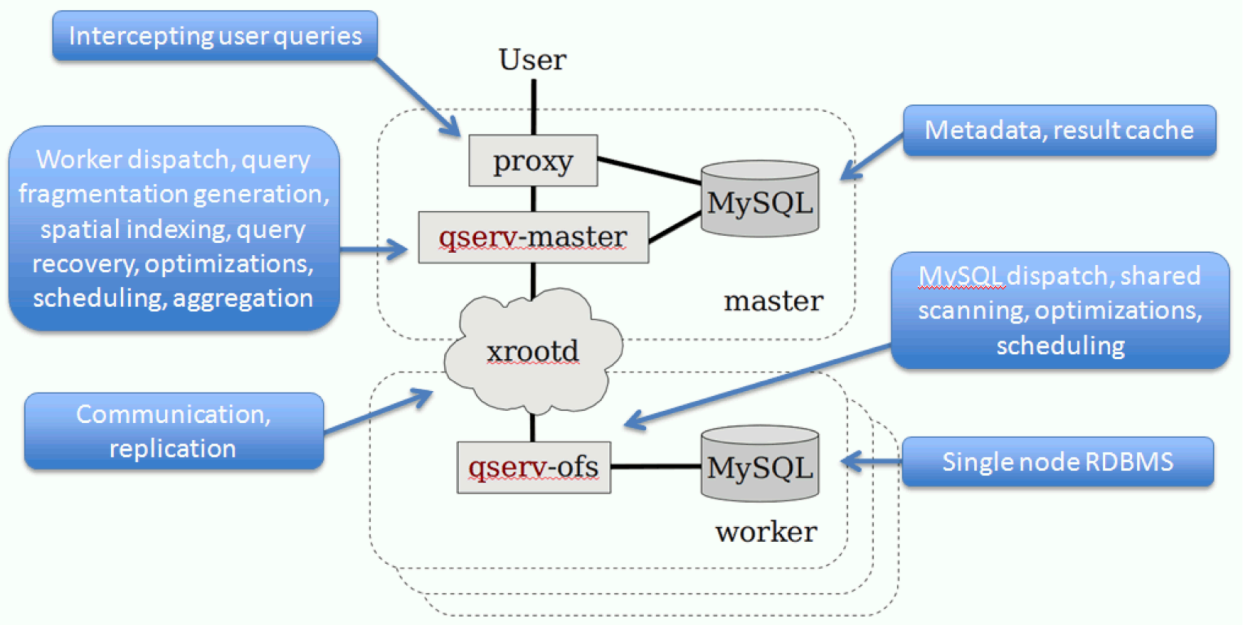

FIGURE 3: XRootD.

While the primary purpose of XRootD is to provide a distributed clustered file system, it is implemented as a plug-in component based system that allows "file" to be replaced by any other resource object. Qserv makes use of this capability to cluster MySQL databases instead of files. Hence, we dispense with calling XRootD a distributed file system and simply call is a generic system for providing named communication paths, clustering, request scheduling, and error recovery.

7.2 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 ϕ (right ascension/α) and θ (declination/δ). 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.

7.3 Query Generation

Qserv is unusual (though not unique) in processing a user query into one or more queries that are subsequently executed on off-the-shelf single-node RDBMS software. This is done in the hopes of providing a distributed parallel query service while avoiding a full re-implementation of common database features. However, we have found that it is necessary to implement a query processing framework much like one found in a more standard database, with the exception that the resulting query plans contain SQL statements as the intermediate language.

A significant amount of query analysis not unlike a database query optimizer is required in order to generate a distributed execution plan that accurately and efficiently executes user queries. Incoming user queries are first parsed into an intermediate representation using a modified SQL92-compliant grammar (Lubos Vnuk's SqlSQL2). The resulting query representation is equivalent to the original user query, and does not include any stateful interpretation, but may not completely reflect the original syntax. The purpose of this representation is to provide a semantic representation that may be operated upon by query analysis and transformation modules without the complexity of a parse tree containing every node in the original EBNF grammar.

Once the representation has been created, the query representation is processed by two sequences of modules. The first sequence operates on the query as a single statement. A transformation step occurs to split the single representation into a "plan" involving multiple phases of execution, one to be executed per-data-chunk, and a one to be executed to combine the distributed results into final user results. The second sequence is applied on this plan to apply the necessary transformations for an accurate result.

We have found that regular expressions and parse element handlers to be insufficient to analyze and manipulate queries for anything beyond the most basic query syntax constructions.

7.3.1 Processing modules

The processing modules perform most of the work in transforming the user query into statements that can produce a faithful result from a Qserv cluster. These include:

- Identify spatial indexing opportunities. This allows Qserv to dispatch spatially-restricted queries on only a subset of the available chunks constituting a table. Spatial restrictions given in Qserv-specific syntax are rewritten as boolean SQL clauses.

- Identify secondary index opportunities. Qserv databases designate one column (more are under consideration) as a key column where its values are guaranteed to exist in one spatial location. Identification allows Qserv to convert point queries on this column into spatial restrictions.
- Identify table joins and generate syntax to perform distributed join results. Qserv primarily supports "near-neighbor" spatial joins for limited distances defined in the partitioning coordinate space. Arbitrary joins between distributed tables are only supported using the key column. Classify queries according to data coverage and table scanning. By identifying tables scanned in a query, Qserv is able to mark queries for execution using shared scanning, which greatly increases efficiency.

7.3.2 Processing module overview

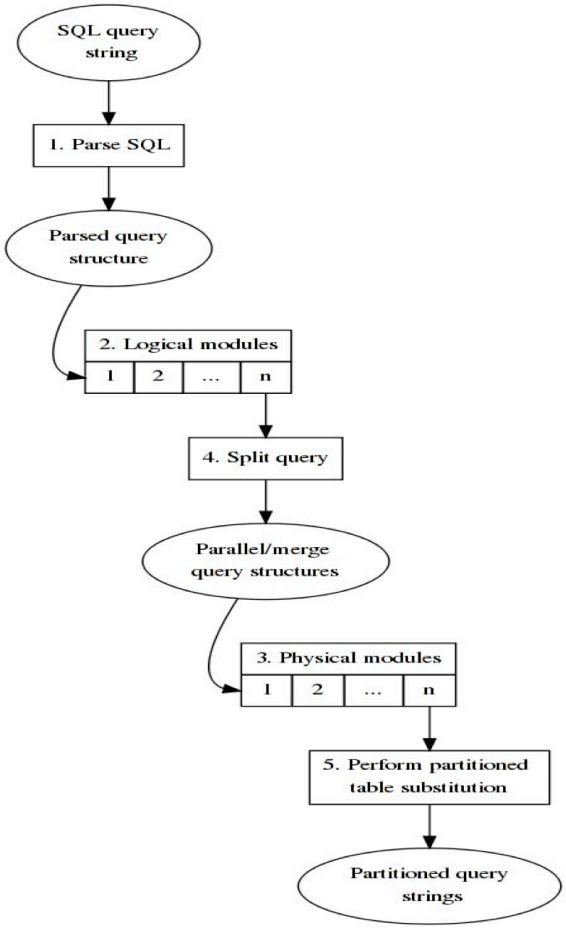

FIGURE 4: Processing modules.

This figure illustrates the query preparation pipeline that generates physical queries from an input query string. User query strings are parsed (1) into a structured query representation that is passed through a sequence of processing modules (2) that operate on that representation in-place. Then, it is broken up (3) into pieces that are explicitly intended for parallel execution on table partitions and pieces intended to merge parallel results into user results. Another processing sequence (4) operates on this new representation, and then finally, concrete query strings are generated (5) for execution.

The two sequences of processing modules provide an extensible means to implement query analysis and manipulation. Earlier prototypes performed analysis and manipulation during parsing, but this led to a practically unmaintainable code base and the functionality has been ported the processing module model. Processing is split into two sequences to provide the flexibility to modules that manipulate the physical structures while offering the simpler single-query representation to modules that do not require the complexity. The clear separation between parsing, whose only goal is to provide a intelligible and modifiable query representation, and the qserv-specific analysis and manipulation is a key factor in the overall flexibility, maintainability, and extensibility of the system and should help the system adapt to current and future LSST needs.

7.4 Dispatch

The baseline Qserv uses XRootD as a distributed, highly-available communications system to allow Qserv frontends to communicate with data workers. Up until 2015, Qserv used a synchronous client API with named files as communication channels. The current baseline system utilizes a general two-way named-channeling system which eliminates explicit file abstractions in favor of generalized protocol messages that can be flexibly streamed. The scheme is called Scalable Service Interface (SSI) and is built on top of XRootD. The interface was specifically designed to hide underlying XRootD dependencies. This allows switching the underlying implementation with minimal impact to Qserv.

7.4.1 Wire protocol

Qserv encodes query dispatches in ProtoBuf messages, which contain SQL statements to be executed by the worker and annotations that describe query dependencies and characteristics. Transmitting query characteristics allows Qserv workers to optimize query execution under changing CPU and disk loads as well as memory considerations. The worker need not re-analyze the query to discover these characteristics or guess at conditions that cannot be

determined by query inspection.

Query results are also returned by ProtoBuf messages. Current versions transmit a MySQL dump file allowing the query results to be faithfully reproduced on the Qserv frontend, but the baseline system will transmit results directly. Initial implementations avoided logic to encode and decode data values, but experience with the prototype MonetDB worker backend proved that data encoding and marshalling were a contained problems whose solution could significantly improve overall query latency by avoiding mutating metadata operations on worker and frontend DBMS systems. Thus the baseline system will encode results in protobuf messages containing schema and row-by-row encoded data values. Streaming results directly from worker dbms instances into frontend dbms instances is a technique under consideration, as is a custom aggregation engine for results that would likely ease the implementation of providing partial query results to end users.

7.4.2 Frontend

ABH In 2012, a new XRootD client API was developed to address our concerns over the older version's scalability (uncovered during a 150 node, 30TB scalability test). The new client API began production use for the broader XRootD community in late 2012. Subsequently, work began under our guidance towards an XRootD Qserv client API that was based on request-response interaction over named channels, instead of opening, reading, and writing files. A production version of this API, the Scalable Service Interface (SSI) became available in early 2015 and Qserv has since been ported to use this interface. The port eliminated a significant body of code that maps dispatching and result-retrieving to file operations. The SSI API will reside in the Xroot code base, where it may be exercised by other projects.

The SSI API provides Qserv with a fully asynchronous interface that eliminates nearly all blocking threads used by the Qserv frontend to communicate with its workers. This eliminated one class of problems we have encountered during large-scale testing. The SSI API has defined interfaces that integrate smoothly with the Protobufs-encoded messages used by Qserv. Two novel features were specifically added to improve Qserv performance. The streaming response interface enables reduced buffering in transmitting query results from a worker mysqld to a the frontend, which lowers end-to-end query latency and reduces storage requirements on workers. The out-of-band meta-data response which arrives prior to the data results can be used to map out the Protobufs encoding and significantly simplify handling response memory buffers.

The fully asynchronous API is crucial on the master because of the large number of concurrent chunk queries in flight expected in normal operation. For example, with the sky split into 10k pieces, having 10 full-scanning queries running concurrently would have 100k concurrent chunk queries–too large a number of threads to allow on a single machine. Hence, an asynchronous API to XRootD is crucial. Threads are used to parallelize multiple CPU-bound tasks. While it does not seem to be important to parse/analyze/manipulate a single user query in parallel (and such a task would be a research topic), the retrieval and processing of results could be done in parallel if some portion of the aggregation/merging were done in Qserv code rather than loaded into the frontend's MySQL instance and merged via SQL queries. Thus results processing should be parallelized among results from individual chunks, and query parsing/analysis/manipulation can be parallelized among independent user queries.

7.4.3 Worker

The Qserv worker uses both threads and asynchronous calls to provide concurrency and parallelism. To service incoming requests from the XRootD API, an asynchronous API is used to receive requests and enqueue them for action. Specifically, the Scalable Service Interface (SSI) is used on Qserv workers as well. The interface provides a mirror image of the actions taken on the front-end making the logic relatively easy to follow and the implementation less error prone.

Threads are maintained in a thread pool to perform incoming queries and wait on calls into the DBMS's API (currently, the apparently synchrnous MySQL C-API). Threads are allowed to run in observance of the amount of parallel resources available. The worker estimates the I/O dependency of each incoming chunk query in terms of the chunk tables involved and disk resources involved, and attempts to ensure that disk access is almost completely sequential. Thus if there are many queries that access the same table chunk, the worker allows as many of them to run as there are CPU cores in the system, but if it has many queries that involve different chunk tables, it allows fewer simultaneous chunk queries in order to ensure that only one table scan per disk spindle occurs. Further discussion of this "shared scanning" feature is described in shared-scans.

7.5 Threading Model

Nearly every portion of Qserv is written using a combination of threaded and asynchronous execution.

Qserv heavily relies on multi-threading to take advantage of all available CPU cores when executing queries, as an example, to complete one full table scan on a table consisting of 1,000 chunks, 1,000 queries (processes) will be executed. To efficiently handle large number of processes that are executed on each worker, we ended up rewriting the XRootD client and switching from thread-per-request model to a thread-pool model. The new client is completely asynchronous, with real call-backs.

mysqlproxy	Single-threaded Lua code
Frontend-python	Single-threaded asynchronous reactor; blocking-thread per user query
Frontend-C++	Processing thread per user-query for preparation; Results-merging thread-per-user-query on-demand;
Frontend-xrootd	Callback threads perform query transmission and results retrieval
Frontend-xrootd internal	Threads for maintaining worker connections (< 1 per host)
Xrootd, cmsd	Small thread pools for managing live network connections and performing lookups
Worker-xrootd plugin	Small thread pool O(#cores) to make blocking mysql C-API calls into local mysqld; callback threads from XRootD perform admission/scheduling of tasks from frontend and transmission of results

7.6 Aggregation

Qserv supports several SQL aggregation functions: AVG(), COUNT(), MAX(), MIN(), and SUM(), and should support SQL92 level GROUP BY.

7.7 Indexing

The secondary index utilizes one or more tables using the InnoDB storage engine on each czar to perform lookups on the database's key colume (objectId). Performance tests (Figure 5) on a single, dual-core host with 1 TB hard disk storage (not SSD) have shown that this configuration will support a full load of 40 billion rows in about 400,000 seconds (110 hours). A more realistic configuration with multiple cores and SSD storage is expected to meet the requirement of fully loading in less than 48 hours.

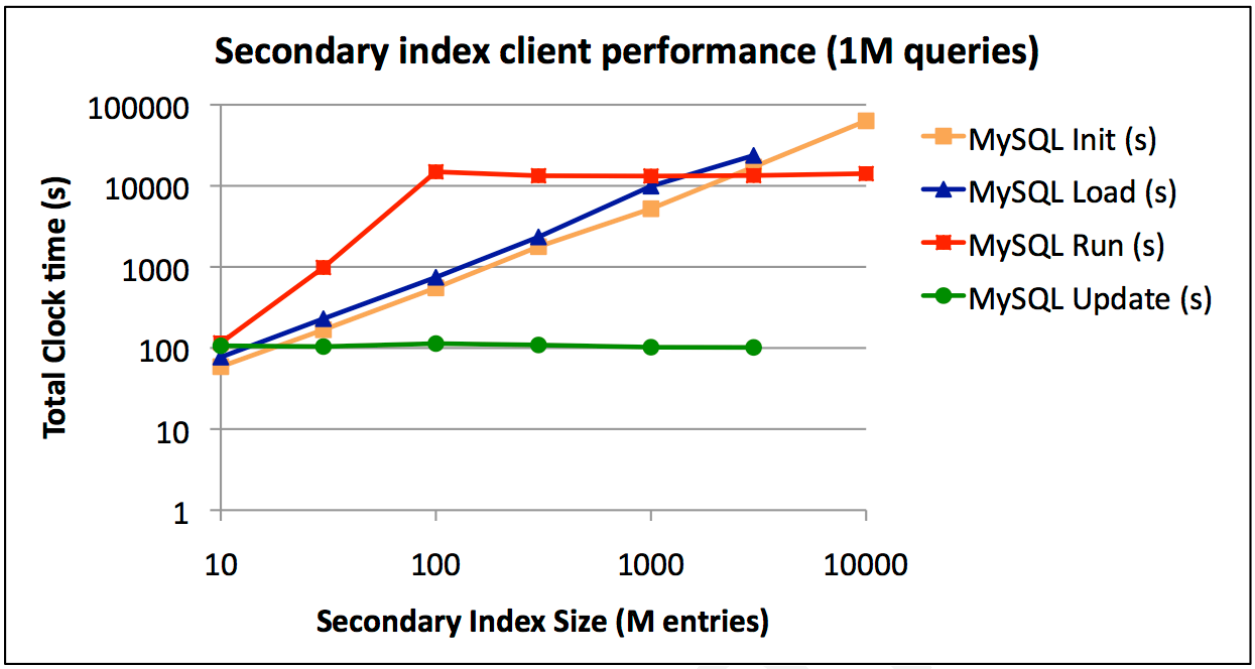

FIGURE 5: Performance tests of MySQL-based secondary index.

To improve the performance of the InnoDB storage engine for queries, the secondary index may be split across a small number (dozens) of tables, each containing a contiguous range of keys. This splitting, if done, will be independent of the partitioning of the database itself. The contiguity of key ranges will allow the secondary index service to identify the appropriate split table arithmetically via an in-memory lookup.

7.7.1 Secondary Index Structure

The secondary index consists of three columns: the key (objectId), the chunk where all data with that key are located (chunkId), and the subchunk within that chunk where data with the key are located (subChunkId). The objectId is assigned by the science pipelines as a 64-bit integer value; the chunkId and subChunkId are both 16-bit integers which identify spatial regions on the sky.

7.7.2 Secondary Index Loading

The InnoDB storage engine loads tables most efficiently if it is provided input data which has been presorted according to the table's primary key. When the secondary index information is collected for loading (from each worker node handling a collection of chunks), it is sorted

by objectId, and may be divided into roughly equal "splits". Each of those splits is loaded into a table *en masse*.

To fully optimize the loading and table splitting, the entire index should be collected from all workers and pre-sorted in memory on the czar. This is not reasonable for 40 billion entries (requiring a minimum of 480 GB memory, plus overhead). Instead, the index data from a single worker can be assumed to be a "representative sample" from the full range of objectIds, so table splitting can be done using the first worker's index data. The remaining workers will be split and loaded according to those defined tables.

7.8 Data Distribution

LSST will maintain the released data store both on tape media and on a database cluster. The tape archive is used for long-term archival. Three copies of the compressed catalog data will be kept. The database cluster will maintain 3 online copies of the data. Because computer clusters of reasonable size failure regularly, the cluster must maintain replicas in order to provide continuous data access. A replication factor of 3 (r=3) is needed in order to determine data integrity by majority rule when one replica is corrupt.

If periodic unplanned downtime is acceptable, an on-tape replica may function as one of the three. However, the use of tape dramatically increases the cost of recovering from a failure. This may be acceptable for some tables, particularly those that are large and lesser-used, although allowing service disruption may make it difficult to make progress on long-running analysis on those large tables.

7.8.1 Database data distribution

The baseline database system will provide access for two database releases: latest and previous . Data for each release will be spread out among all nodes in the cluster.

Data releases are partitioned spatially, and spatial pieces (chunks) are distributed in a round-robin fashion across all nodes. This means that area queries involving multiple chunks are almost guaranteed to involve resources on multiple nodes.

Each node should maintain at least 20% free space of its data storage allocation. The remaining free space is then available to be "borrowed" when another node fails. This will a temporary use of storage capacity until more server resources can be put online, until the

80% storage use is returned.

7.8.2 Failure and integrity maintenance

There will be failures in any large cluster of node, in the nodes themselves, in data storage volumes, in networks access and so on. These failures will remove access to data that is resident on those nodes, but this loss of data access should not affect that ability of scientists to analyze the dataset as a whole. We need to set a data availability time over 99.5% to ensure confidence of the community in the stability of the system. To ensure this level of data access, and to allow acceptable levels of node failures in a cluster, there will be replication of data on a table level throughout the cluster.

The replication level will be that each table in the database will exist 3 times, each on separate nodes. A monitoring layer to the system will check on the availability of each table every few hours, although this time will be tuned in practice. When this layer sees that a table has less than three replicas available, this will initiate a replication of that table to another nodes, not currently hosting that table. The times for the checking, and speed of replication will be tuned to the stability of the cluster, such that about 5% of all tables at any given time will only have 1 or 2 replicas. Three replicas will ensure that tables will be available even in cases of large failures, or when nodes need to be migrated to new hardware in bulk.

Should an entire node fail, replicating that data to another single node would be fairly expensive in terms of time. As of July 2013, a 3TB drive will have a write speed of 60/150/100 MB/s (min/max/avg) [12] and refilling this single drive would remove access to that replica of the data for about 8 hours. We plan on having free space on each node, and only fill local storage to 80%. The free space will be used for temporary storage of tables on failures, where replicas can take place in parallel between nodes into this free space. When new nodes with free storage are added to the cluster, then this data can be copied off this free space into the drive, taking the full 8 hours, but there will still be 3 replicas of data during this time. Once this is complete, this data will have 4 replicas for the short period of time while these tables can be removed from the temporary storage, returning each node to 80% usage.

7.9 Metadata

Qserv needs to track various metadata information, static (not changing or changing very infrequently), and dynamic (run-time) in order to maintain schema and data integrity and optimize the cluster usage.

7.9.1 Static metadata

Qserv typically works with databases and tables distributed across many different machines: it breaks individual large tables into smaller chunks and distributes them across many nodes. All chunks that belong to the same logical table must have the same schema and partitioning parameters. Different tables often need to be partitioned differently, for example some tables might be partitioned with overlap (such as the Object table), some might be partitioned with no overlap (for example the Source table), and some might not need partitioning at all (e.g., a tiny Filter table). Further, there might be different partitioning strategies, such as spherical-box based, or HTM-based. All this information about schema and partitioning for all Qserv-managed databases and tables needs to be tracked and kept consistent across the entire Qserv cluster.

Implementation of the static metadata in Qserv is based on hierarchical key-value storage which uses a regular MySQL database as a storage backend. This database is shared between multiple masters and it must be served by a fault-taulerant MySQL server instance, e.g. using a master-master replication solution like MariaDB Galera cluster. Database consistency is critical for metadata and it should be implemented using one of the transactional database engines in MySQL.

Static metadata may contain following information:

- Per-database and per-table partitioning and scan scheduling parameters.
- Table schema for each table, used to create database tables in all worker and master instances; the schema in the master MySQL instance can be used to obtain the same information when a table is already created.
- Database and table state information, used primarily by the process of database and table creation or deletion.
- Definitions for the set of worker and master nodes in a cluster including their availability status.

The main clients of the static metadata are:

- Administration tools (command-line utilities and modules) which allow one to define or modify metadata structures.

- Qserv master(s), mostly querying partitioning parameters but also allowed to modify table/database status when deleting/creating new tables and databases. Master(s) should not depend on node definitions in metadata, the xrootd facility is used to communicate with workers.
- Special "watcher" service which implements distributed process of database and table management.
- An initial implementation of the data loading application which will use the node definitions and will create/update database and table definitions. This initial implementation will eventually be replaced by a distributed loading mechanism which may be based on separate mechanisms.

7.9.2 Dynamic metadata

In addition to static metadata, a Qserv cluster also needs to track its current state, and keep various statistics about query execution. This sort of data is udated frequently, several times per query execution, and is called dynamic metadata.

Prototype implementation of the dynamic metadata is based on MySQL database. Like static metadata it needs to be shared between all master instances and will be served via a single fault-talerant MySQL instance which will be shared with static metadata database.

Dynamic metadata will contain the following information:

- Definition of every master instance in a Qserv cluster.
- Record of every SELECT-type query processed by cluster. This record includes query processing state and some statistical/timing information.
- Per-query list of table names used by the asynchronous queries, this information is used to delay table deletion while async queries are in progress.
- Per-query worker information, which includes chunk id and identifying information for the worker processing that chunk id. This information will be used to transparently restart the master or migrate query processing to a different master in case of master failure.

The most significant use of the dynamic metadata is to track execution of asyncronous queries. When an async query is submitted it is registered in dynamic metadata and its ID is returned to the user immediately. Later users can request status information for that query ID which is obtained from dynamic metadata. When query processing is finished users can request results from that query, and the master can obtain the location of the result data from dynamic metadata.

Additionally dynamic metadata can be used to collect statistical information about queries that were executed in the past which may be an important tool in understanding and improving system performance.

7.9.3 Architecture

The Qserv metadata system is implemented based on master/server architecture: the metadata is centrally managed by a *Qserv Metadata Server* (qms). The information kept on each worker is kept to a bare minimum: each worker only knows which databases it is supposed to handle, all remaining information can be fetched from the qms (through Qserv) as needed. This follows our philosophy of keeping the workers as simple as possible.

The real-time metadata is managed inside qms in in-memory tables, periodically synchronized with disk-based table. Such configuration allows reducing qms latency—important to avoid delaying query execution time. Should a qms failure occur, the in-flight queries for which the information was lost will be restarted. Since the synchronization to disk-based table will occur relatively frequently (eg. at least 1 per minute), the lost time is insignificant. To avoid overloading the qms with, only the high-level information available from Qserv-master is stored in qms; all worker-based information is cached in a scratch space locally to each worker in a simple, raw form (e.g, key-value, ASCII file), and can be fetched on demand as needed.

At the moment we use xml-rpc as a message protocol to communicate with qms. It was a natural choice given that this protocol is already in use by Qserv master.

7.9.4 Typical Data Flow

Static metadata:

1. Parts of the static metadata known before data is partitioned/loaded are loaded by the administration scripts responsible for loading data into the database, then these scripts

start data partitioner.

2. The data partitioner reads static metadata loaded by the administration scripts, loads remaining information.
3. When Qserv starts, it fetches all static metadata and caches it in memory in a special, in-memory optimized C++ structure.
4. The contents of the in-memory metadata cache inside Qserv can be refreshed on demand if the static metadata changes (for example, when a new database or a table is added).

Dynamic-metadata:

1. Master loads the information for each query (when it starts, when it completes).
2. Detailed statistics are dumped by each worker into a scratch space kept locally. This information can be requested from each worker on demand. A typical use case: if all chunk-queries except one completed, qms would fetch statistics for the still-running chunk-query to estimate when the query might finish, whether to restart this query etc.

7.10 Shared Scans

Arbitrary full-table scanning queries must be supported in LSST's baseline catalog, and in order to provide this support cost-effectively and efficiently, Qserv implements shared scans. Shared scans effectively reduces the I/O cost of executing multiple scanning queries concurrently, reducing the system hardware need and purchasing costs.

Shared scans reduce overall I/O costs by forcing incoming queries to share. When multiple queries scan the same table, theoretically, they can completely share I/O and incur only the I/O cost of a single query rather than the sum of their individual costs. In general, it is difficult for queries to share I/O because their arrival times are random and uncorrelated. Each query begins scanning at different times, and because LSST's catalog tables will be so large, general system caching is ineffective. In Qserv, scanning queries are broken up into many parts, and shared scanning forces each query to operate on the same portion and thus share I/O cost, rather than allowing each to perform its own ordered scan and incur costs individually.

7.10.1 Background

Historically, shared scanning has been a research topic that has very few real-world implementations. We know of only one implementation in use (Teradata). Most database implementations assume OS or database caching is sufficient, encouraging heavy use of indexing to reduce the need of table scans. However, our experiments have shown that when tables are large enough (by row count) and column access sufficiently variable (cannot index enough columns when there are hundreds to choose from), indexes are insufficient. With large tables, indexes no longer fit in memory, and even when they do fit in memory, the seek cost to retrieve each row is dominant when the index selects a percentage of rows, rather than some finite number (thousands or less).

7.10.2 Implementation

The implementation of shared scans in Qserv is in two parts. The first part is a basic classification of incoming queries as scanning queries or non-scanning queries. A query is considered to scan a table if it depends on non-indexed column values and involves more than k chunks (where k is a tunable constant). Note that involving multiple chunks implies that the query selects from at least one partitioned table. This classification is performed during query analysis on the front-end and leveraging table metadata. The metadata includes a "scan rating", which is set by hand. Higher scan ratings indicate larger tables that take longer to read from disk. The identified "scan tables" and their ratings are marked and passed along to Qserv workers, which use the information in scheduling the fragments of these scanning queries.

The second part of the shared scans implementation is a scheduling algorithm that orders query fragment execution to optimize cache effectiveness. Because Qserv relies on individual off-the-shelf DBMS instances on worker nodes, it is not allowed to modify those instances to implement shared scans. Instead, it issues query fragments ordered to maximize locality of access in data and time, and tries to lock the files associated with the tables in memory as much as possible. Using the identified scan tables and their ratings, the worker places them on the appropriate scheduler. There will be at least three schedulers. One for queries expected to complete in under an hour, which are expected to be related to the Object table. One for queries expected to take less than eight hours, expected to be related to Object_Extra. And one for scans expected to take eight to twelve hours for ForcedSource and/or Source tables. The reasoning being that a single slow query can impede the progress of a shared scan and all the other user queries on that scan. There may be a need for another scheduler to handle queries taking more than 12 hours.

Each scheduler places incoming chunk queries into one of two priority queues sorted by chunk id then scan rating of the individual tables. If the query is for a chunk after the currently scanning chunk id, it is placed on the active priority queue, otherwise it is placed on the pending priority queue. After chunk id, the priority queue is sorted by the table with highest scan rating to ensure that the largest tables in the chunk are grouped together.

Once the query is on the appropriate scheduler, the algorithm proceeds as follows. When a dispatch slot is available, it checks the highest priority scheduler. If that scheduler has a query fragment, hereafter called tasks, and it is not at its quota limit, it is allowed to start its next task, otherwise the worker checks the next scheduler. It continues doing this until a task has been started or all the schedulers have been checked.

Each scheduler is only allowed to start a task under certain circumstances. There must be enough threads available from the pool so that none of the other schedulers are starved for threads as well as enough memory available to lock all the tables for the task in memory. If the scheduler has no tasks running, it may start one task and have memory reserved for the tables in that task. This should prevent any scheduler from hanging due to memory starvation without requiring complicated logic but could incur extra disk I/O. More on locking tables in memory later.

Schedulers check for tasks by first checking the top of the active priority queue. If the active priority queue is empty, and the pending priority queue is not, then the active and pending queues are swapped with the task being taken from the top of the “new” active queue.

Since the queries are being run by a separate DBMS instance of which there is little control of how it goes about running queries, the worker can control when queries are sent to the DBMS and also lock files in memory. Files in memory are among the most likely items to be paged out when memory resources are low, which would increase disk I/O. Locking files in memory prevents this from happening. However, care must taken in choosing how much memory can be used for locking files. Use too much and there will be a significant impact on DBMS performance. Set aside too little, and schedulers will not make optimum use of the resources available and may be forced to run tasks without actually locking the files in memory.

The memory manager controls which files are locked in memory. When a scheduler tries to run a task, the task asks the memory manager to lock all the shared scan tables it needs. The memory manager determines which files are associated with the tables. If the files are already locked in memory and there is enough memory available to lock the files which are

not already locked, the task is given a handle and allowed to run. When the task completes, it hands the handle back to the memory manager. If it was the last task using any particular table, the memory for the files used by that table is freed.

When the memory manager locks a file, it does not read the file. It only sets aside memory for the file to occupy when it is read by the DBMS. In the special case where a task can run even though there is not enough memory available, those tables that cannot fit are put on a list of reserved tables and their size is subtracted from the quota until they can be locked or freed. When memory is freed, the memory manager will try to lock the reserved tables.

Because Qserv processes interactive, short queries concurrently with scanning queries, its query scheduler should be able to allow for those queries to complete without waiting for a query scan. To achieve this, Qserv worker nodes choose between the scan scheduler described above and a simpler *grouping* scheduler. Incoming queries with identified scan tables are admitted to the scan scheduler, and all other queries are admitted to the grouping scheduler. The grouping scheduler is a simple scheduler that is a simple variant of a plain FIFO (first-in-first-out) scheduler. Like a FIFO scheduler, it maintains a queue of queries to execute, and operates identically to a FIFO scheduler with one exception–queries are grouped by chunk id. Each incoming query is inserted into the queue behind another query on the same chunk, and at the back if no queued query matches. The grouping scheduler assumes that the queue will never get very long, because it is intended to only handle short interactive queries lasting fractions of seconds, but groups its queue according to chunk id in order to provide a minimal amount of access locality to improve throughput at a limited cost to latency. Some longer queries will be admitted to the grouping scheduler even though they are scanning queries, provided that they have been determined to only scan a single chunk. Although these non-shared scan query will disrupt performance of the overall scan on the particular disk on a worker, the impact is thought to be small because each of these represents all (or a large fraction of) the work for a single user query, and the impact is amortized among all disks on all workers.

For discussion about the performance of the existing prototype, refer to demo-shared-scans.

7.10.3 Memory management

To minimize system paging when multiple threads are scanning the same table, we implemented a memory manager called memman. When a shared scan is about to commence, the shared scan scheduler informs memman about the tables the query will be using and

how important it is to keep those tables in memory during the course of the query. When directed to keep the tables in memory, memman opens each data base table file, maps it into memory, and then locks the pages to prevent the kernel from stealing the pages for other uses. Thus, once a file page is faulted in, it stays in memory and allows other threads to scan the contents of the page without incurring additional page faults. Once the shared scan of the table completes, memman is told that the tables no longer need to remain in memory. memman frees up the pages by unlocking them and deleting the mapping.

This type of management is necessary to satisfy system paging requirements because the prime paging pool is the set of unlocked file system pages.

7.10.4 XRootD scheduling support

When the front-end dispatches a query, the XRootD normally picks the least used server in an attempt to spread the load across all of the nodes holding the required table. While this works well for interactive queries, it is hardly ideal for shared scan queries. In order to optimize memory and I/O usage, queries for the same table in a shared scan should all be targeted to the same node. A new scheduling mode was added to the XRootD cmsd called affinity scheduling. The front-end can tell XRootD whether or not a particular query has affinity to other queries using the same table. Queries that have affinity are always sent to the same node relative to the table they will be using. This allows the back-end scheduler to minimize paging by running the maximum number of queries against the same table in parallel. Should that node fail, XRootD assigns another working node that has the table as the target node for queries that have affinity.

7.10.5 Multiple tables support

Handling multiple tables in shared scans requires an additional level of management. The scheduler will aim to satisfy a throughput yielding average scan latencies as follows:

- `Object` queries: 1 hour
- `Object`, `Source` queries (join): 12 hours
- `Object`, `ForcedSource` queries (join): 12 hours
- `Object_Extras`[10] queries (join): 8 hours.

[10] This includes all `Object`-related tables, e.g., `Object_Extra`, `Object_Periodic`, `Object_NonPeriodic`, `Object_APMean`

As stated in 8.10.2, there will be schedulers for queries that are expected to take one hour, eight hours, or twelve hours. The schedulers group the the tasks by chunk id and then the highest scan rating of the all tables in the task. The scan ratings are meant to be unique per table and indicative of the size of the table, so that this sorting places scans using the largest table from the same chunk next to each other in the queue. Using scan rating allows flexibility to work with data sets with schemas different than that of LSST.

Since scans are not limited to specific tables, complicated joins could occur in user queries that could take more than twelve hours to process. The worker may also need to be able to identify user queries that are too slow for the current scheduler based on the time it takes to complete tasks for that query. This indicates there may be a need for a scheduler to handle queries with very long run times.

7.11 Level 3: User Tables, External Data

Level 3 tables including tables generated by users, and data catalogs brought from outside, depending on their type and size, will be either partitioned and distributed across the production database servers, or kept unpartitioned in one central location. While the partitioned and distributed Level 3 data will share the nodes with Level 2 data, it will be kept on dedicated disks, independent from the disks serving Level 2 data. This will simplify maintenance and recoverability from failures.

Level 3 tables will be tracked and managed through the Qserv Metadata System (qms), described in Section 7.9. This includes both the static, as well as the dynamic metadata.

7.12 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 Scalable Service Interface (SSI) 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.

Cluster management performed outside of XRootD does not directly affect query execution, but include coordinating data distribution, loading, nodes joining/leaving and is discussed in qserve-admin. The SSI API includes methods that allow dynamic updates to the data view

of an XRootD cluster. So that when new tables appear or disappear, the XRootD system will incorporate that information for future scheduling decisions. Thus, clusters can dynamically change without the need to restart the XRootD system.

7.13 Fault Tolerance

Qserv approaches fault tolerance in several ways. The design exploits the immutability of the underlying data by replicating and distributing data chunks across a cluster such that in the event of a node failure, the problem can be isolated and all subsequent queries re-routed to nodes maintaining duplicate data. Moreover, this architecture is fundamental to Qserv's incremental scalability and parallel performance. Within individual nodes, Qserv is highly modularized with minimal interdependence among its components, which are connected via narrow interfaces. Finally, individual components contain specialized logic for minimizing, handling, and recovering from errors.

The components that comprise Qserv include features that independently provide failure-prevention and failure-recovery capabilities. The MySQL proxy is designed to balance its load among several underlying MySQL servers and provide automatic fail-over in the event a server fails. The XRootD system provides multiple managers and highly redundant servers to provide high bandwidth, contend with high request rates, and cope with unreliable hardware. And the Qserv master itself contains logic that works in conjunction with XRootD to isolate and recover from worker-level failures.

A worker-level failure denotes any failure mode that can be confined to one or more worker nodes. In principle, all such failures are recoverable given the problem nodes are identified and alternative nodes containing duplicate data are available. Examples of such failures include a disk failure, a worker process or machine crashing, or network problems that render a worker unreachable.

Consider the event of a disk failure. Qserv's worker logic is not equipped to manage such a failure on localized regions of disk and would behave as if a software fault had occurred. The worker process would therefore crash and all chunk queries belonging to that worker would be lost. The in-flight queries on its local mysqld would be cleaned up and have resources freed. The Qserv master's requests to retrieve these chunk queries via XRootD would then return an error code. The master responds by re-initializing the chunk queries and re-submits them to XRootD. Ideally, duplicate data associated with the chunk queries exists on other nodes. In this case, XRootD silently re-routes the request(s) to the surviving node(s) and all associated

queries are completed as usual. In the event that duplicate data does not exist for one or more chunk queries, XRootD would again return an error code. The master will re-initialize and re-submit a chunk query a fixed number of times (determined by a parameter within Qserv) before giving up, logging information about the failure, and returning an error message to the user in response to the associated query.

Error handling in the event that an arbitrary hardware or software bug (perhaps within the Qserv worker itself) causes a worker process or machine to crash proceeds in the same manner described above. The same is true in the event that network loss or transient sluggishness/overload has the limited effect of preventing XRootD from communicating with one or more worker nodes. As long as such failures are limited to a finite number of workers and do not extend to the Qserv master node, XRootD is designed to record the failure and return an error code. Moreover, if duplicate data exists on other nodes, this will be registered within XRootD, which will successfully route any subsequent chunk queries.

In the event of an unrecoverable error, the Qserv master is equipped with a status/error messaging mechanism designed to both log detailed information about the failure and to return a human-readable error message to the user. This mechanism includes C++ exception handling logic that encapsulates all of the master's interactions with XRootD. If an unrecoverable exception occurs, the master gracefully terminates the query, frees associated resources, logs the event, and notifies the user. Qserv's internal status/error messaging system also generates a status message and timestamp each time an individual chunk query achieves a milestone. Such milestones include: chunk query dispatch, written to XRootD, results read from XRootD, results merged, and query finalized. This real-time status information provides useful context in the event of an unrecoverable error.

Building upon the existing fault-tolerance and error handling features described above, future work includes introducing a heart-beat mechanism on worker nodes that periodically pings the worker process and will restart it in the event it becomes unresponsive. Similarly, a master monitoring process could periodically ping worker nodes and restart a worker machine if necessary. We are also considering managing failure at a per-disk level, but this would require research since application-level treatment of disk failure is relatively rare. It should also be possible to develop an interface for checking the real-time status of queries currently being processed by Qserv by leveraging its internally used status/error messaging mechanism.

7.14 Next-to-database Processing

We expect some data analyses will be very difficult, or even impossible to express through SQL language. This might be particularly useful for time-series analysis. For this type of analyses, we will allow users to execute their analysis algorithms in a procedural language, such as Python. To do that, we will allow users to run their own code on their own hardware resources co-located with production database servers. Users then run queries on the production database which stream rows directly from database cluster nodes to the user processing cluster, where arbitrary code may run without endangering the production database. This allows their incurred database I/O needs to be satisfied using the database system's shared scanning infrastructure while providing the full flexibility of running arbitrary code.

7.15 Administration

7.15.1 Installation

Qserv as a service requires a number of components that all need to be running, and configured together. On the master node we require mysqld, mysql-proxy, XRootD, cmsd, qserv metadata service, and the qserv master process. On each of the worker nodes there will also be the mysqld, cmsd, and XRootD service. These major components come from the MySQL, XRootD, and Qserv distributions. But to get these to work together we will also require many more software package, such as protobuf, lua, expat, libevent, python, zope, boost, java, antlr, and so on. And many of these require more recent versions than you are provided in most system distributions. We have an installation layer, developed by SLAC, and LPC in Clermont-Ferrand, France in collaboration, which will determine the packages, configure, compile and install them in an automated process.

Currently, the Qserv installation procedure supports only the official LSST platform— RHEL6, and SL6 Linux distributions. Other UNIX-like systems will be supported in the future as needed. The Qserv package first can be downloaded from SLAC for install. In the initial README there are basic install procedures, which start with a bootstrap script, that will perform a yum install of needed packages distributed with RHEL6, where the versions will support the Qserv install. Once that is done an install script can be started. This will first download needed packages not shipped with RHEL6 from SLAC, and get those installed first. All software will be installed into a sandbox root path, and all installed by the production username. Along with this is an install of MySQL from source that will be configured for Qserv. These further packages will be configured to run together, and then Qserv will be complied and linked to these in-

stalled packages. All this runs without user interaction, and usually completes within 15 to 20 minutes, to provide a complete Qserv either master or worker node.

7.15.2 Data loading

As previously mentioned, Data Release Production will not write directly to the database. Instead, the DRP pipelines will produce binary FITS tables and image files that are reliably archived as they are produced. Data will be loaded into Qserv in bulk for every table, so that tables are either not available, or complete and immutable from the user query access perspective.

For replicated tables, these FITS files are converted to CSV (e.g. by harvesting FITS image header keyword value pairs, or by translating binary tables to ASCII), and the resulting CSV files are loaded directly into MySQL and indexed. For partitioned tables like Object and Source, FITS tables are fed to the Qserv partitioner, which assigns partitions based on sky coordinates and converts to CSV.

In particular, the partitioner divides the celestial sphere into latitude angle "stripes" of fixed height H. For each stripe, a width W is computed such that any two points in the stripe with longitudes separated by at least W have angular separation of at least H. The stripe is then broken into an integral number of chunks of width at least W, so that each stripe contains a varying number of chunks (e.g. polar stripes will contain just a single chunk). Chunk area varies by a factor of about pi over the sphere. The same procedure is used to obtain subchunks: each stripe is broken into a configurable number of equal-height "substripes", and each substripe is broken into equal-width subchunks. This scheme is preferred over the Hierarchical Triangular Mesh for its speed (no trigonometry is required to locate the partition of a point given in spherical coordinates), simplicity of implementation, and the relatively fine control it offers over the area of chunks and sub-chunks.

The boundaries of subchunks constructed as described are boxes in longitude and latitude - the overlap region for a subchunk is defined as the spherical box containing all points outside the subchunk but within the overlap radius of its boundary.

The task of the partitioner is to find the IDs of the chunk and subchunk containing the partitioning position of each row, and to store each row in the output CSV file corresponding to its chunk. If the partitioning parameters include overlap, then the row's partitioning position might additionally fall inside the overlap regions of one or more subchunks. In this case, a

copy of the row is stored for each such subchunk (in overlap CSV files).

Tables that are partitioned in Qserv must be partitioned identically within a Qserv database. This means that chunk tables in a database share identical partition boundaries and identical mappings of chunk id to spatial partition. In order to facilitate table joining, a single table's columns are chosen to define the partitioning space and all partitioned tables (within a related set of tables) are either partitioned according that pair of columns, or not partitioned at all. Our current plan chooses the Object table's `ra_PS` and `decl_PS` columns, meaning that rows in the Source and ForcedSource tables will be partitioned according to the Objects they reference.

There is one exception: we allow for pre-computed spatial match tables. As an example, such a table might provide a many-to-many relationship between the LSST Object catalog and a reference catalog from another survey, listing all pairs of LSST Objects and reference objects separated by less than some fixed angle. The reference catalog cannot be partitioned by associated Object, as more than one Object might be matched to a reference object. Instead, the reference catalog must be partitioned by reference object position. This means that a row in the match table might refer to an Object and reference object assigned to different chunks stored on different Qserv worker nodes.

We avoid this complication by again exploiting overlap. We mandate (and verify at partitioning time) that no match pair is separated by more than the overlap radius. When partitioning match tables, we store a copy of each match in the chunk of both positions referenced by that match. When joining Objects to reference objects via the match table then, we are guaranteed to find all matches to Objects in chunk C by joining with all match records in C and all reference objects in C or in the overlap region of C.

All Qserv worker nodes will partition subsets of the pipeline output files in parallel – we expect partitioning to achieve similar aggregate I/O rates to those of full table scans for user query access, so that partitioning should complete in a low factor (2-3x) of the table scan time. Once it does, each Qserv worker will gather all output CSV files for its chunks and load them into MySQL. The structure of the resulting chunk tables is then optimized to maximize performance of user query access (chunk tables will likely be sorted, and will certainly be compressed), and appropriate indexes are built. Since chunks are sized to fit in memory, all of these steps can be performed using an in-memory file-system. I/O costs are incurred only when reading the CSV files during the load and when copying finalized tables (i.e. .MYD/.MYI files) to local disk.

The last phase of data loading is to replicate each chunk to one other Qserv worker node. We will rely on table checksum verification rather than a majority rule to determine whether a replica is corrupt or not.

The partitioner has been prototyped as a multi-threaded C++ program. It uses an in-memory map-reduce implementation internally to scale across cores, and can read blocks of one or more input CSV files in parallel. It does not currently understand FITS table files. CSV file writes are also parallelized - each output chunk is processed by a single reducer thread and can be written to in parallel with no application level locking. In preliminary testing, our partitioner was able to sustain several hundred MB/s of both read and write bandwidth when processing a CSV dump of the PT1.2 Source table.

We are investigating a pair of data loading optimizations. One is to have pipeline processes either integrate the partitioning code or feed data directly to the partitioner, rather than communicating via persistent storage. The other is to write out tables in the native database format (e.g. as .MYD files, ideally using the MySQL/MariaDB server code to do so), allowing the CSV database loading step to be bypassed.

7.15.3 Administrative scripts

The administration of the qserv cluster will require a set of scripts, all run from the one master machine, to control the large set of workers. The main admin script, qserv-admin, will supply the base needs, with starting all processes needed for the service, in order, and taking down all processes to stop the service. Also base monitoring of service is supplied here, to report on processes that are running, and responding to base queries, to check on MySQL or XRootD dying or locking up. Also is supplied is the updating of the configuration definitions from the master out to all workers, such that all machines need to have the same configurations for the services.

The base data loading onto the nodes tends to be a slightly detailed process, beyond the just the data preparation. Up to now, data preparation produces text files in csv format, and then these will be loaded into the MySQL layer as a MyISAM table. The schema for these tables will need to have added to them the fields for chunk and subchunk number needed for the Qserv service. The modification of the schema and the control of the loading of the data, which can take hours, is done with the qserv-load script. The loading of the data is also done without index creation, and then that is done after the data loading. We are also experimenting with the use of compressed read-only tables for the data serving, and this is an option.

Another needed setup for the data service in qserv, is the creation of the "emptyChunks" list. The data will be spatially partitioned into "chunks", as previously described, but the for the complete service, the master process with need to know how many chunks exist in the data, and which of these chunks contain no data. In queries which will involve a complete table scan, which chunks to create query, or not, will need to be known. Once the data is loaded, there is a another script which will go out to nodes and see what chunks are there, and compile a list of all possible chunks and which chunks do not contain data, or the "emptyChunks" lists. This is loaded by the qserv master process at startup.

7.16 Result Correctness

To verify Qserv does not introduce any unexpectedly alter results (e.g., does not show the same object twice or does not miss any objects on the chunk boundaries), we developed an automated testbed, which allows us to run pre-set queries on pre-set data sets both through plain MySQL and through Qserv, and compare results.

7.17 Current Status and Future Plans

As of now (June 2013) we have implemented a basic version of the system end-to-end. Our prototype is capable of parsing a wide range of queries, including queries executed by our QA system, "PipeQA", rewriting them into sub-queries, executing these sub-queries in parallel and returning results to the user. The implementation includes a generic parser, basic query scheduler, job executor, query result collector. We demonstrated running all query types (low, high, super-high such as large-area near-neighbor) including aggregations, scalably on a 150-node cluster using 30 TB data set; and a smaller subset of queries scalably on 300-node cluster (remaining tests in progress, expecting to complete in the next 2-3 weeks) we also demonstrated the system performs well enough to meet the LSST query response time requirements. We demonstrated the system can handle high-level of concurrency (10 concurrent queries simultaneously accessing 10,000 chunks each). We demonstrated the system can recover from a variety of faults, or at minimum gracefully fail if the error is unrecoverable. We extended SQL syntax coverage and ensured the system is capable of supporting all types of queries executed over the course of recent data challenges by PipeQA and users. We implemented a core foundation for the metadata, currently used for managing static metadata about Qserv-managed databases and tables, a set of administrative tools, and scalable data partitioner. We made the system easy to set up, resilient to typical failures and common user mistakes. We implemented automated test bed. We consider the current prototype to have

a quality of a typical late-alpha / early-beta software.

Future work includes:

- extending metadata to support run-time statistics, implementing query management tools
- implementing support for Level 3 data
- completing initial shared scan implementation, testing and implementing concurrent and synchronized shared scans on multiple spindles
- demonstrating cross-match with external catalogs
- improving interfaces for users (eg hiding internal tables)
- re-examining and improving query coverage, including more advanced SQL syntax, such as sub-queries as needed
- improvements to administration scripts
- support for HTM partitioning in Qserv
- authentication and authorization
- resource management
- early partition results
- performance improvements
- partition granularity varying per table
- security

Extending metadata to support run-time statistics, implementing query management tools. Qserv currently does not maintain any explicit run-time system state. Keeping such state would simplify managing Qserv cluster, and building features such as query management: currently there are no tools for inspecting and managing queries in-flight, and there are no interfaces for halting queries except upon error detection. It is clear that users and administrators will need to list running queries, check query status and possibly abort queries.

Implementing support for Level 3 data. Qserv will need to support level 3. That means users should be able 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.

Completing initial shared scan implementation, testing and implementing concurrent and synchronized shared scans on multiple spindles. The first prototype implementation of shared scanning is mostly complete, with the remaining work focused on basic analysis and characterization of incoming user queries to determine scanning tables and plumbing to convey the appropriate hints to worker nodes.

Demonstrating cross-match with external catalogs. One of the use cases involves 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.

Improving interfaces for users. Many admin-type commands such as "list processes" or "explain" are not ported to the distributed Qserv architecture, and thus will not show correct result. At the moment we have disabled these commands. Additionally, commands such as listing tables in a given database will have to be overloaded, for example, we should show user a table "Object" (even though in practice such table does not exist in the Qserv system), instead of all the chunk `Object_XXX` tables, that are internal, and should not be exposed to the end-user.

Re-examining and improving query coverage, including more advanced SQL syntax, such as sub-queries as needed. We examined what queries users and production processes execute, however we realize this query set is far from the complete list of queries we will see in the future. All needed syntax needs to be understood and fully supported. 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.

Improvements to administration scripts. To further automate common tasks related to database management, table management, partition management, data distribution, and others we need to implement many improvements to the administration scripts.

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.

Security. The system needs to be secure and resilient against denial of service attacks.

7.18 Open Issues

What follows is a (non-exhaustive) list of issues, technical and scientific, that are still being discussed and where changes are possible.

- **Support for updates**. Size of Level 1 catalog is relatively small, and the expected query access patterns are relatively non-challenging, thus currently do not envision any need to deploy scalable Qserv-like architecture for Alert Production. Should this change, we will need to support updates in Qserv, which will likely have some non-trivial impact on the architecture.
- **Very large objects**. Some objects (eg, large galaxies) are much larger than our overlap region, in some cases their footprint will span multiple chunks. Currently we are working with the object center, neglecting the actual footprint. While there are some science use cases that would benefit from a system that tracks objects based on their footprint, this is currently not a requirement. Potential solution would involve adding a custom index similar to the r-tree-based indexes such as the TOUCH [15].
- **Very large results**. Currently, the front-end that disptached the query is responsible for assembling the results. In general, this is not a scalable approach as the resources required to processes the results may be several orders of magnitude greater than those needed to dispatch the query. One solution is to replicate the front-end to the extent necessary to handle query results. Alternatively, the Scalable Service Interface can be augmented to allow running disconnected queries. That is, once a particular front-end dispatches a query it can get a handle to that query and disconnect from it. Another server can, using that handle, reconnect to the query and process the results. This is a more flexible model as it allows independent scaling of query dispatch and result processing. It also has the aded benefit of not cancelling in-progress queries dispatched by a particulr fron-end should that front-end die.

8 Large-scale Testing

8.1 Introduction

8.1.1 Ideal environment

Based on the detailed spreadsheet analysis, we expect the ultimate LSST production system will be composed of few hundred database servers [LDM-144], so a realistic test should include a cluster of at least 100 nodes.

Total database size of a single data release will vary from ~1.3 PB (DR1) to ~15 PB (DR11).[11] 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 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

[11] These numbers are for single copy, data and indices, compressed when appropriate.

architecture system and turning MySQL.

8.1.2 Schedule of testing

- Selecting the base technology – Q2 2009
- Determining the architecture – 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) – Q4 2010
- Scalability and performance improvements and tests on a large cluster (150-250 nodes) – Q4 2011
- Large scale tests, performance tests on a large cluster (250-300 nodes) – Q2 2013
- Shared scans – Q3 2013
- Fault tolerance – Q4 2013

8.1.3 Current status of tests

We have run several large scale tests

1. (10/2009) 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)
2. (2010) several 100-node tests run at SLAC [16]. These tests helped us uncover many bottlenecks and prompted rewriting parts of our software, as well as implementing several missing features in XRootD.
3. (4/2011) A 30 TB test on 150-node SLAC cluster using Qserv in 40/100/150 node configurations, using 2 billion row Object and 32 billion row Source tables, total of 30 TB data set.
4. (12/2012) A 100-terabyte test on JHU's *Datascope* 20-node cluster. Due to high instability of the cluster this test turned into testing resilience to faults of Qserv and the associated administrative tools.

5. (05/2013) A 10,000-chunk test on a 120-node SLAC cluster to reproduce and understand subtle issues with concurrency at scale.
6. (07/2013) A test on 300-node IN2P3 cluster to re-test scalability, performance, concurrency, and uncover unexpected issues and bottlenecks.
7. (08/2013) A demonstration of shared scans.

The tests 3-6 listed above are described in further details below.

8.2 150-node Scalability Test

8.2.1 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.

8.2.2 Data

We tested using a dataset synthesized by spatially replicating the dataset from the LSST data challenge ("PT1.1"). We used two tables: Object and Source.[12] 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).[13] 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

[12] The schema may be browsed online at `http://lsst1.ncsa.uiuc.edu/schema/index.php?sVer=PT1_1`

[13] Source for the duplicator is available at `https://github.com/lsst/qserv/blob/master/admin/python/lsst/qserv/admin/dataDuplicator.py`

~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).

8.2.3 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*).

```
SELECT * FROM Object WHERE objectId = <objId>
```

In fig-150-node-low-vol-object-retrieval 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.

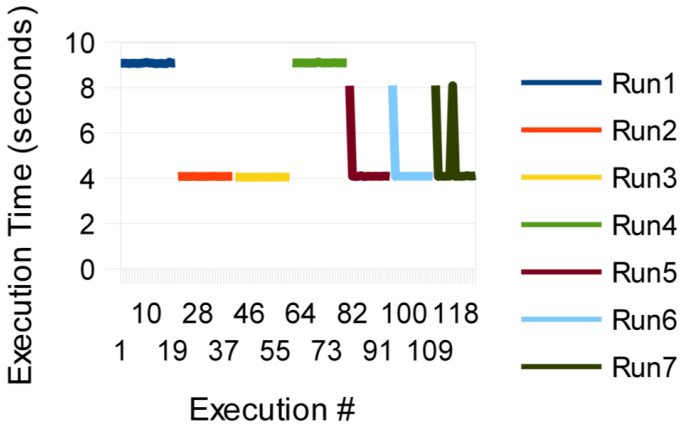

FIGURE 6: Low-volume object retrival.

```
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 fig-low-volume-time-series 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.

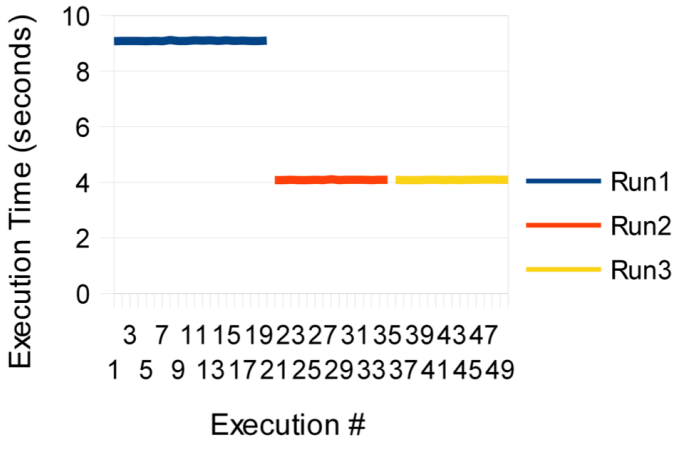

FIGURE 7: Low-volume time series.

```
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;
```

In fig-low-volume-spatial-filter 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.

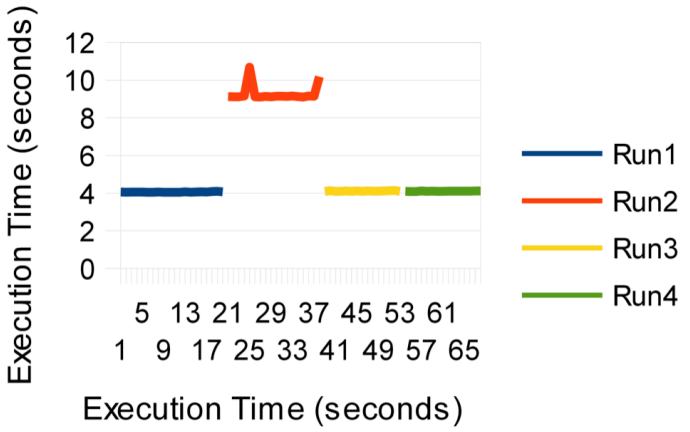

FIGURE 8: Low-volume spatially-restricted filter.

```
SELECT COUNT(*) FROM Object
```

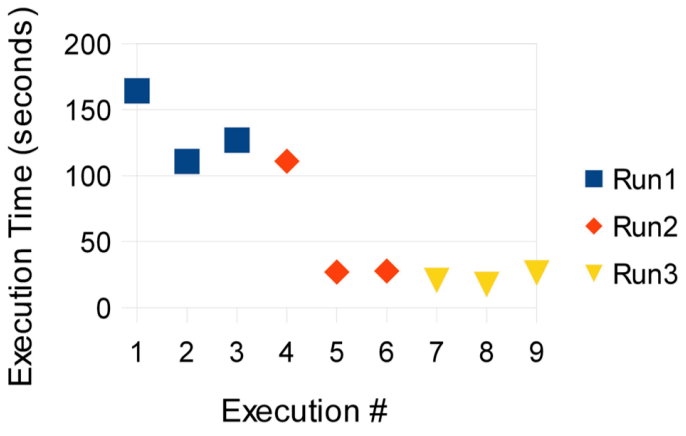

FIGURE 9: High volume count.

```
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
```

Using the on-disk data footprint (MySQL's MyISAM .MYD, without indexes or metadata) of the Object table (1.824×10^{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.

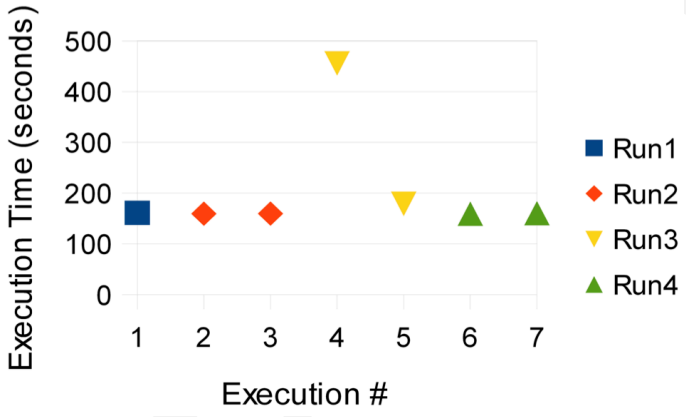

FIGURE 10: High volume full-sky filter.

```
SELECT COUNT(*) AS n, AVG(ra_PS), AVG(decl_PS), chunkId
FROM Object
GROUP BY chunkId
```

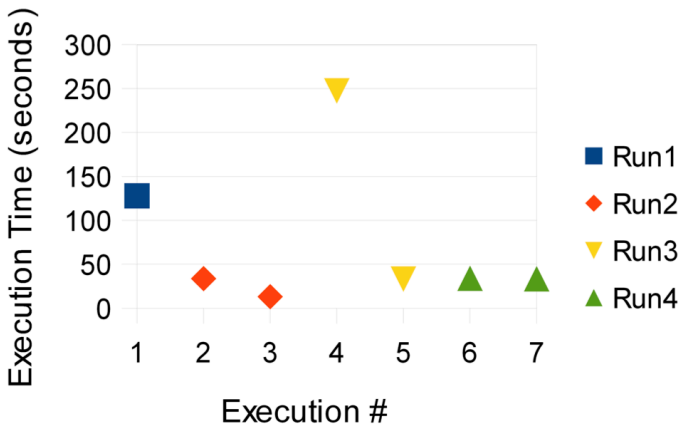

FIGURE 11: High volume full-sky filter.

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 fig-150-node-high-volume-density 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.

```
SELECT COUNT(*)
FROM Object o1, Object o2
WHERE qserv_areaspec_box(-5,-5,5,-5)
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.

```
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 $k \sim 41$. 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.

8.2.4 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.

8.2.4.1 Scaling with small queries From fig-150-node-scaling-small-1 — fig-150-node-scaling-small-3, 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 fig-150-node-scaling-small-3 is caused by 2 slow queries (23s and 57s); the other 28 executed in times ranging from 4.09 to 4.11 seconds.

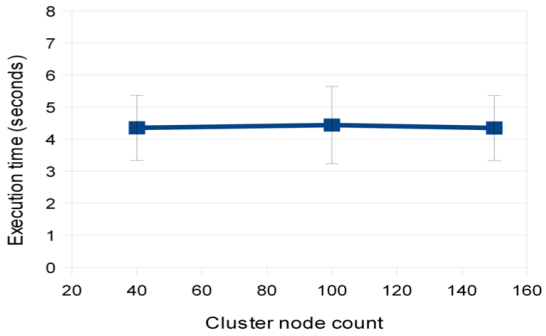

FIGURE 12: Scaling with node count (1).

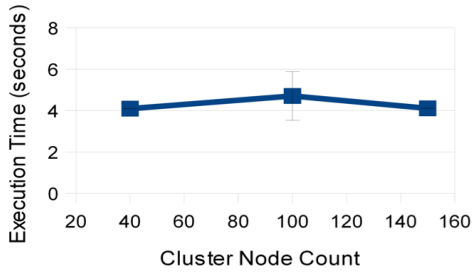

FIGURE 13: Scaling with node count (2).

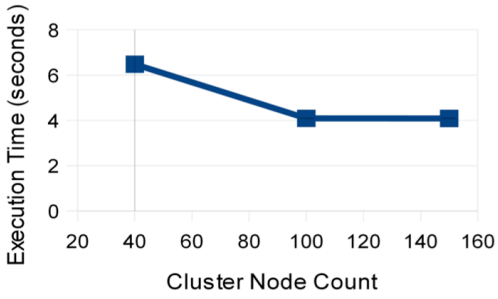

FIGURE 14: Scaling with node count (3).

8.2.4.2 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 fig-150-node-scaling-high-volume 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.

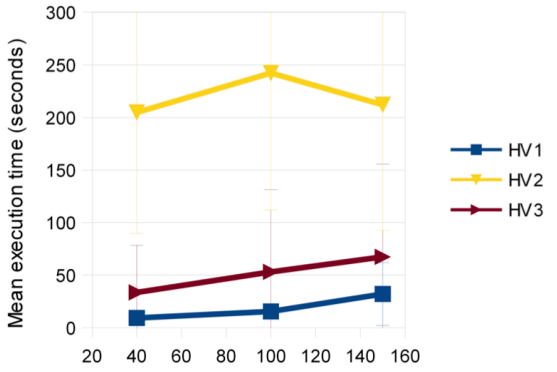

FIGURE 15: Scaling with high volume queries.

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.

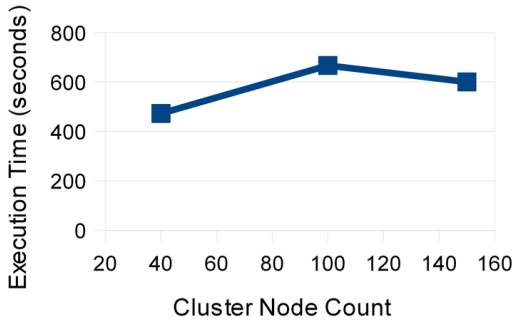

FIGURE 16: Scaling with super high volume queries.

8.2.5 Concurrency

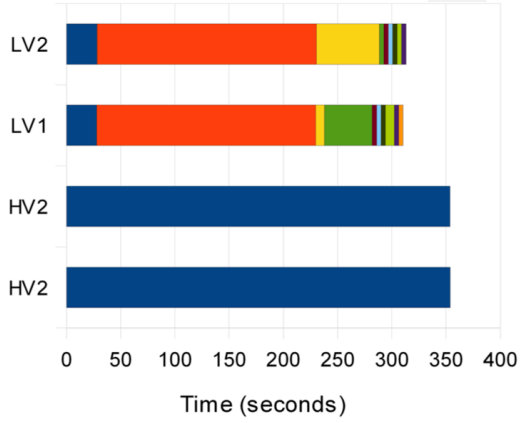

FIGURE 17: Concurrency test.

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.

8.2.6 Discussion

8.2.6.1 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.

8.2.6.2 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).

8.2.6.3 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.

8.2.6.4 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 [13] 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.

8.2.6.5 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.

8.3 100-TB Scalability Test (JHU 20-node cluster)

In the fall of 2012, we were provided the opportunity to use a cluster of computers at John Hopkins University (JHU), that were large memory, multi-processor computers, each with very large storage attached, to setup as a Qserv service. There were 21 nodes provided for us, and the nodes had two processors with 12 cores each of Intel Xeon X5650 CPUs at 2.67GHz. But mounted as a data volume, were 22TB raid arrays, to provide a possible 450TB of storage. This would provide a high volume storage test, but with a low number of compute nodes, with one master node, and 20 worker nodes.

The data used would be produced on each node, starting with a test dataset called "pt12". This test dataset was 220GB, but high density data in one particular spot in the sky, a few degrees wide. We chose a high density spot of this data, and then duplicated with across the whole sphere, to provide a high density dataset over the whole sky, yielding an estimated 100TB of data.

The production of this large amount of data proved to have problems. The production was rather slow, taking many weeks for a full production on each node. At first long processes were setup, and the stability of the cluster was an issue, with processes dying after days of running, and many smaller production processes were setup to get past stability issues. This produced over 100TB of csv text files, into about 7000 chunks worth of data. Once that was done, then this data was loaded into MySQL MyISAM tables, and with the large data sizes this also took days.

Over the course of this time, often nodes would go out with problems, and be down for some amount of time, before coming back. Often this would be with the data still on the mounted volumes, but the loss of computing nodes would set back the time until the data would be complete. But this was a small problem, than the problem of data stability. Once all the nodes were up and running, the data service was still having problems. This was found to be either loss or corruption of a few of the thousands of tables on the various nodes. With loss of data, either data would be re-created or just blocked for testing. Over the course of testing, dealing with data corruption was constant issue, but still a large percentage would be accessible at least. Also a problem was file corruption on the install software, and a few nodes needed to be reinstalled over the course of the testing. A full 100TB of data was generated, but only about 85TB could ever be served, before the resources had to be given back.

But some testing was able to get done on this cluster. A full table scan was performed on the Object data, although this was only on the aprox, 85TB of data was served, not the complete 100TB of data that was generated. The query was performed:

```
SELECT count(*) FROM Object
```

```
+------------+
| count(*)   |
+------------+
| 2059335968 |
+------------+
```

```
1 row in set (19.26 sec)
```

Showing 2B objects in the table. The result here was after the query was performed a few times, and the cache had been stabilized. This is similar to the times found from previous testing.

The low volume test from access to a small portion of the sky was also performed. Using the query:

```
SELECT count(*)
FROM Object
WHERE qserv_areaspec_box(1,3,2,4)
AND scisql_fluxToAbMag(zFlux_PS) BETWEEN 21 AND 21.5
```

```
+----------+
| count(*) |
+----------+
| 748      |
+----------+

1 row in set (4.45 sec)
```

This time for access is also similar to previous testing, looking for the number of object in a small part of the sky of a certain color. The ~4.5 sec. overhead here is a baseline overhead for data access in this version of the qserv software.

A high volume data test was performed, looking for color information on records within a certain range of color. This will scan over all objects, to return certain number of records. The test query was:

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

This query returned 15695 records in 6 min 33.50 sec. Again this query was performed a number of times, and this time is the average time after the caches had stabilized. This query was performed again, this time looking at a lower number of records, looking for the difference between *i* and *z* flux of 5 this time. This query returned 2967 records, in 6 min 14.0 sec. The time was a little lower this time, which was mostly the time to print the records to the screen, where the rest of the time was the over-head in scanning the available object data to return these records. The previous tests were done on 30TB of data, but using 150 nodes, although these nodes had many less cores. But this test would return in about 180 sec there, where here it is about 375 sec. The extra time here will come from the access of larger amounts of data per node, and amount of data in general, and the access rate of the data storage.

8.4 Concurrency Tests (SLAC 100,000 chunk-queries)

A previous version of the Qserv master code had dedicated two fixed size thread pools to each query, one for dispatching chunk queries, and the other for reading back results. The dispatch pool was sized at a quite high 500 threads for two reasons. Firstly, one goal was to dispatch work as quickly as possible, allowing the Qserv workers to prioritize as they know best. Secondly, the first query dispatch against a chunk takes ~5s, so that cluster cold start latency on a full table scan of ~10,000 chunks takes approximately 100s with this many dispatch threads. Subsequently, XRootD caching allows for near instantaneous dispatches in comparison.

The thread pool for result reads was given a much smaller size: just 20 threads per query. This is because the Qserv master process can only exploit limited amounts of parallelism when merging worker results for a single query. In fact, the main benefit of using a number as high as 20 threads is that it reduces result merge latency when chunk query execution times (and hence result availability times) are skewed.

An unfortunate consequence of this simple design was that running too many concurrent queries would cause thread creation failures in the Qserv master process. We therefore changed to a unified query dispatch and result reader thread pool model.

To test our ability to handle many concurrent full-table scan queries without running out of threads, we partitioned the PT1.2 Object table into ~8,000 chunks, and distributed them across an 80 node cluster at SLAC. The nodes in this cluster were quite old and had limited quantities of RAM, making them the perfect workhorses for this sort of test. In particular, asking for more than ~1000 threads would cause Qserv master failure on this platform. Using the new unified thread-pool design we were able to successfully run between 2 and 12 concur-

rent Object table scans each involving ~8,000 chunk queries, requiring a total execution time of 2 to 8 minutes, thus demonstrating that the Qserv master can handle loads of ~100,000 in-flight chunk queries, even on very old hardware.

Note that using a unified thread pool for result reads requires special measures to avoid query starvation. A single query can easily require 10,000 result reads and there will be far fewer total threads in the pool. As a result, we must be careful to avoid assigning all threads to a single query, or queries that should be interactive can easily become decidedly non-interactive as they wait for a table scan to finish. Our unified thread pool implementation therefore assigns available threads to the query using the fewest worker threads, and makes sure to create new threads when a new query is encountered (up to some hard limit).

To test this, we setup Qserv with a single master node and a single worker node. The worker was configured with ~12,000 empty chunk tables. We then submitted both full-table scans (`SELECT COUNT(*) FROM Object`), and an interactive query (`SELECT COUNT(*) FROM Object WHERE objectId IN (1)`) requiring just a single chunk query to answer. Though we were able to demonstrate that the master immediately allocated threads to dispatch the lone interactive chunk query and read back its results, the execution time of the interactive query was still far higher than it should have been. It turns out this is because the Qserv worker uses a FIFO chunk query scheduling policy, and the single chunk query corresponding to the interactive user query was being queued up behind a multitude of chunk queries from the full table scan on the worker side. We are currently working to address this deficiency as part of ongoing work on shared scans.

8.5 300-node Scalability Test (IN2P3 300-node cluster)

The largest test we run to-date was run during July-September of 2013 on a 300-node cluster at the IN2P3 center in France. The main purpose of the test was to test Qserv scalability and performance beyond 150 nodes, and re-check concurrency at scale.

8.5.1 Hardware

Test machines were quad-core Intel(R) Xeon L5430 running at 2.66GHz speed, with small local spinning disks (120GB of usable space), and 16GB of RAM per machine. We received an allocation of 320 nodes, which was intended to allow for a 300 node test in spite of some failed nodes (and indeed, 17 nodes failed during the tests!)

8.5.2 Data

With only 120GB of available storage per node, only a limited amount of data could get produced. We tuned our data synthesizer to produce 50GB of table data per node, giving 15TB of aggregate data. 220GB of LSST PT1.2 data was synthesized into a dense stripe covering the declination range -9˚ to +9˚, setting the partitioning to 120 stripes and 9 substripes (1.5˚ x 1.5˚ chunks and 10 arc-minute-sided subchunks). This yielded 3,000 chunks of data, with 9 to 11 chunks of data on each node. The Object table had 0.4 billion rows, and the Source table had 14 billion rows. Data partitions for the Source table averaged 4GB.

Data synthesis took a couple hours for the Object table and overnight for the Source table. Recent work on the installation software enabled data loading ingest to happen within a couple hours (comparing to ~6 days for the 150-node test we run two years earlier.)

8.5.3 Software stability issues identified

The initial Qserv installation did not function for queries involving 300 nodes, even though subsets involving 10, 50, 100, and 150 nodes functioned properly. The first culprit was the use of an older XRootD release that was missing recent patches for a particular client race condition. Another culprit was instability exacerbated by excessive use of threads in the original threading model that the testing in section 9.5 was to address. This was addressed by re-tuning relevant threading constants. The new XRootD client has alleviated this problem.

8.5.4 Queries

```
SELECT * FROM Object WHERE objectId = <id>
```

End-to-end user execution time averaged at 1.1 seconds.

```
SELECT count(*)
FROM Object
WHERE qserv_areaspec_box(1,3,2,4) AND
      scisql_fluxToAbMag(zFlux_PS) BETWEEN 21 AND 21.5
```

This average query response time was 1.3 sec. This is roughly the minimum end-to-end execution time for query that selects small region for this version of Qserv.

```
SELECT count(*) FROM Object
WHERE qserv_areaspec_box(1,2,3,4)
AND scisql_fluxToAbMag(zFlux_PS) BETWEEN 21 AND 21.5
AND scisql_fluxToAbMag(gFlux_PS)–
    scisql_fluxToAbMag(rFlux_PS) BETWEEN 0.3 AND 0.4
AND scisql_fluxToAbMag(iFlux_PS)–
    scisql_fluxToAbMag(zFlux_PS) BETWEEN 0.1 AND 0.12
```

This average query response time was 1.3 sec. The extra CPU expense of the conditions was insignificant.

```
SELECT s.ra, s.decl
FROM Object o
JOIN Source s USING (objectId)
WHERE o.objectId = 142367243760566706
AND o.latestObsTime = s.taiMidPoint
```

This returned in an average time of 11.2 sec.

```
SELECT COUNT(*) FROM Object
```

End-to-end user execution time averaged 7.8 seconds. This is the minimum overhead to dispatch queries for all 3,000 chunks to all 300 nodes and retrieve their results. A condition-less `COUNT(*)` is executed as a metadata lookup by MySQL when using MyISAM tables, involving almost no disk I/O.

Similar query executed on the Source table returned in 11.9 seconds.

```
SELECT COUNT(*) FROM LSST.Object WHERE gFlux_PS>1e-25
```

This query was repeated with different constants in the filtering condition, and the execution time did not vary significantly – it returned in an average time of 8.45 sec – or less than 1 second longer than the condition-less `COUNT(*)` query.

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

Varying the flux difference filter in a range of 4-5, the execution time ranged between 7-9 seconds.

```
SELECT objectId, ra_PS, decl_PS,
       scisql_fluxToAbMag(zFlux_PS)
FROM LSST.Object
WHERE scisql_fluxToAbMag(zFlux_PS) BETWEEN 25 AND 26
```

End-to-end execution time ranged from 7.7 to 8.4 seconds.

```
SELECT objectId
FROM Source
JOIN Object USING(objectId)
WHERE qserv_areaspec_box(1,3,2,4)
```

This returned in 9 min 42.9 sec. Some portion of this time was spent printing the results to the screen (this test utilized a standard MySQL command-line client).

```
SELECT COUNT(*) FROM Object o1, Object o2
WHERE qserv_areaspec_box(-5,-5,5,5)
AND scisql_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 large part of the sky (100 deg^2 area). The execution times was 4 min 50 sec. The resultant row counts was ~7 billion.

8.5.5 Scaling

We run a subset of the above queries on different number nodes (50, 100, 250, 200, 250, 300), in "week scaling" configuration, to determine how our software scales.

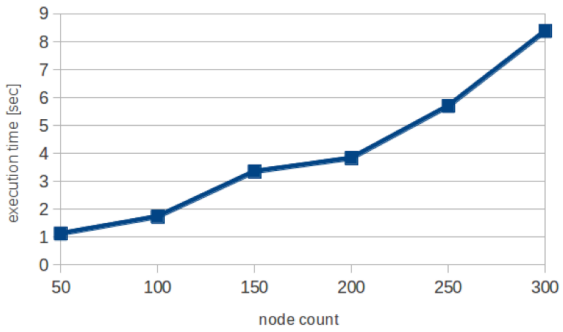

FIGURE 18: Dispatch overhead.

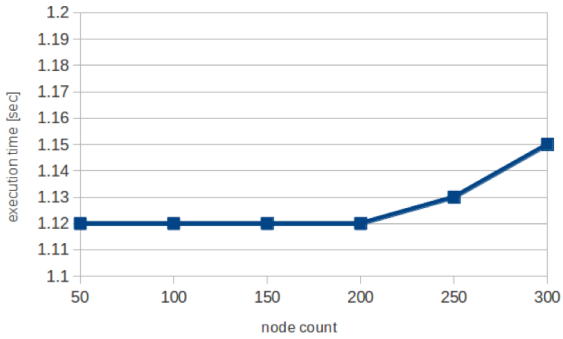

FIGURE 19: Simple object selection.

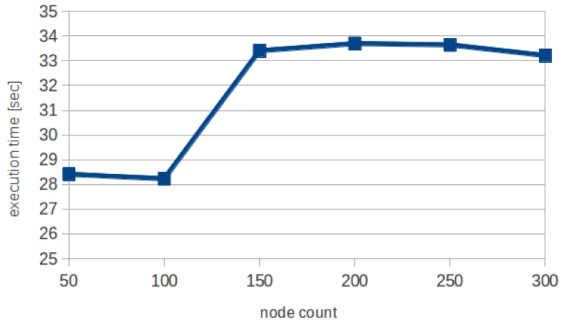

FIGURE 20: Select from a mid-size area.

8.5.6 Discussion

We showed linear scalability of the dispatch – see fig-in2p3-dispatch-overhead, achieving below 10 sec (12 for Source catalog) times when run on the entire, 300 node cluster. Queries that touch all chunks on all clusters are required to complete under an hour, so 10-12 sec overhead is very low. During previous large scale tests we run on 150 nodes 2 years ago, we were getting ~4 sec overhead. During this test, we measured 3.3 sec on 150-node configuration, which indicates we reduced the overhead, however since hardware used for these two tests was not the same, direct comparison would not be entirely fair.

We showed the overhead for simple, interactive queries was on the order of 1.8 sec when dispatching a query on one of the 300-nodes (see fig-in2p3-simple-object-selection). Yes, we can observe a non linearity starting from ~200 nodes, however that non-linearity is on the order of 0.03 second when going from 200 to 300 nodes. Since we are required to answer interactive queries under 10 sec, the <20% overhead is already acceptable, though we are planning to reduce it further in the future.

We were able to run all interactive-type queries well under required 10 second, with the exception of simple Object-Source join, which tool 11.2 sec. The longer-than-ideal time is attributed to unnecessary materialization of subchunks for every query that involves a join – this is expected to be optimized and alleviated in the near future.

More complex queries, such as a query that selects from a mid-size region showed linear scalability as well (fig-in2p3-mid-size-area). The one time 6-sec “jump” between 100 and 150

node test is attributed to switching to different number of chunks: as we reduced the size of the cluster from 150 to 100 nodes, we excluded some chunks that were previously falling inside searched region.

We were also able to run complex queries, such as full table scans and near neighbor queries, and did not observe any anomalies.

It is important to note that due to the ratio of data size to RAM, a large fraction of the data, in particular for the "small" Object catalog was cached in memory. Such environment was particularly good for testing dispatch and result returning overheads, however it would be unfair to approximate observed performance to production-size data sets, especially given that we also had a smaller number of chunks (3,000 in the test vs expected 20,000 in production).

9 Other Demonstrations

9.1 Shared Scans

We have conducted preliminary empirical evaluation of our basic shared scan implementation. The software worked exactly as expected, and we have not discovered any unforeseen challenges. For the tests we used a mix of queries with a variety of filters, different CPU load, different result sizes, some with grouping, some with aggregations, some with complex math. Specifically, we have measured the following:

- A single full table scan through the Object table took ~3 minutes. Running a mix 30 such queries using our shared scan code took 5 min 27 sec (instead of expected ~1.5 hour it'd take if we didn't use the shared scan code.)
- A single full table scan through Source table took between ~14 min 26 sec and 14 min 36 sec depending on query complexity. Running a mix of 30 such queries using shares scan code took 25 min 30 sec. (instead of over 7 hours).

In both cases the extra time it took comparing to the timing of a single query was related to (expected) CPU contention: we have run 30 simultaneous queries on a slow, 4-core machine.

In addition, we demonstrated running simultaneously a shared scan plus short, interactive queries. The interactive queries completed as expected, in some cases with a small (1-2 sec) delay.

9.2 Fault Tolerance

To prove Qserv can gracefully handle faults, we artificially triggered different error conditions, such as corrupting random parts of a internal MySQL files while Qserv is reading them, or corrupting data sent between various components of the Qserv (e.g., from the XRootD to the master process).

9.2.1 Worker failure

These tests are meant to simulate worker failure in general, including spontaneous termination of a worker process and/or inability to communicate with a worker node.

When a relevant worker (i.e. one managing relevant data) has failed prior to query execution, either 1) duplicate data exists on another worker node, in which case XRootD silently routes requests from the master to this other node, or 2) the data is unavailable elsewhere, in which case XRootD returns an error code in response to the master's request to open for write. The former scenario has been successfully demonstrated during multi-node cluster tests. In the latter scenario, Qserv gracefully terminates the query and returns an error to the user. The error handling of the latter scenario involves recently developed logic and has been successfully demonstrated on a single-node, multi-worker process setup.

Worker failure during query execution can, in principle, have several manifestations.

1. If XRootD returns an error to the Qserv master in response to a request to open for write, Qserv will repeat request for open a fixed number (e.g. 5) of times. This has been demonstrated.
2. If XRootD returns an error to the Qserv master in response to a write, Qserv immedi-

ately terminates the query gracefully and returns an error to the user. This has been demonstrated. Note that this may be considered acceptable behavior (as opposed to attempting to recover from the error) since it is an unlikely failure-mode.

3. If XRootD returns an error to the Qserv master in response to a request to open for read, Qserv will attempt to recover by re-initializing the associated chunk query in preparation for a subsequent write. This is considered the most likely manifestation of worker failure and has been successfully demonstrated on a single-node, multi-worker process setup.

4. If XRootD returns an error to the Qserv master in response to a read, Qserv immediately terminates the query gracefully and returns an error to the user. This has been demonstrated. Note that this may be considered acceptable behavior (as opposed to attempting to recover from the error) since it is an unlikely failure-mode.

9.2.2 Data corruption

These tests are meant to simulate data corruption that might occur on disk, during disk I/O, or during communication over the network. We simulate these scenarios in one of two ways. 1) Truncate data read via XRootD by the Qserv master to an arbitrary length. 2) Randomly choose a single byte within a data stream read via XRootD and change it to a random value. The first test necessarily triggers an exception within Qserv. Qserv responds by gracefully terminating the query and returning an error message to the user indicating the point of failure (e.g. failed while merging query results). The second test intermittently triggers an exception depending on which portion of the query result is corrupted. This is to be expected since Qserv verifies the format but not the content of query results. Importantly, for all tests, regardless of which portion of the query result was corrupted, the error was isolated to the present query and Qserv remained stable.

9.2.3 Future tests

Much of the Qserv-specific fault tolerance logic was recently developed and requires additional testing. In particular, all worker failure simulations described above must be replicated within a multi-cluster setup.

9.3 Multiple Qserv Installations on a Single Machine

Once in operations, it will be important to allow multiple qserv instances to coexist on a single machine. This will be necessary when deploying new Data Release, or for testing new version of the software (e.g., MySQL, or Qserv). In the short term, it is useful for shared code development and testing on a limited number of development machines we have access to. We have successfully demonstrated Qserv have no architectural issues or hardcoded values such as ports or paths that would prevent us from running multiple instances on a single machine.

10 References

[1] **[Document-1386]**, Becla, J., 2006, *Database Ingest Tests*, Document-1386, URL `https://ls.st/Document-1386`

[2] Becla, J., 2012, Spatial Join Performance, URL `http://dev.lsstcorp.org/trac/wiki/db/SpatialJoinPerf`

[3] Becla, J., 2013, Queries Used for Scalability & Performance Tests, URL `https://dev.lsstcorp.org/trac/wiki/db/queries/ForPerfTest`

[4] Becla, J., Lim, K.T., 2013, Common Queries, URL `https://dev.lsstcorp.org/trac/wiki/db/queries`

[5] **[LDM-141]**, Becla, J., Lim, K.T., 2013, *Data Management Storage Sizing and I/O Model*, LDM-141, URL `https://ls.st/LDM-141`

[6] Becla, J., Lim, K.T., Monkewitz, S., Nieto-Santisteban, M., Thakar, A., 2008, In: Argyle, R.W., Bunclark, P.S., Lewis, J.R. (eds.) Astronomical Data Analysis Software and Systems XVII, vol. 394 of Astronomical Society of the Pacific Conference Series, 114, ADS Link

[7] **[Document-11701]**, Becla, J., Lim, K.T., Wang, D., 2011, *Evaluation of Solid State Disks*, Document-11701, URL `https://ls.st/Document-11701`

[8] **[DMTN-046]**, Becla, J., Lim, K.T., Wang, D., 2013, *An investigation of database technologies*, DMTN-046, URL https://dmtn-046.lsst.io, LSST Data Management Technical Note

[9] Dorigo, A., Elmer, P., Furano, F., Hanushevsky, A., 2005, WSEAS Transactions on Computers, 4, 348, URL http://xrootd.org/presentations/xpaper3_cut_journal.pdf

[10] **[LDM-144]**, Freemon, M., Pietrowicz, S., Alt, J., 2016, *Site Specific Infrastructure Estimation Model*, LDM-144, URL https://ls.st/LDM-144

[11] **[Document-7025]**, Kantor, J., Krabbendam, V., 2011, *DM Risk Register*, Document-7025, URL https://ls.st/Document-7025

[12] Kirsch, N., 2012, WD Red 3TB NAS Hard Drive Review, URL http://www.legitreviews.com/article/2092/3/

[13] Kunszt, P.Z., Szalay, A.S., Thakar, A.R., 2001, In: Banday, A.J., Zaroubi, S., Bartelmann, M. (eds.) Mining the Sky, 631, doi:10.1007/10849171_83, ADS Link

[14] Monash, C., 2010, eBay followup — Greenplum out, Teradata > 10 petabytes, Hadoop has some value, and more, URL http://www.dbms2.com/2010/10/06/ebay-followup-greenplum-out-teradata-10-petabytes-hadoop-has-some-value-and-more/

[15] Nobari, S., Tauheed, F., Heinis, T., et al., 2013, In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, 701–712, ACM, New York, NY, USA, doi:10.1145/2463676.2463700

[16] Wang, D., Becla, J., 2012, Phase II Qserv Testing (up to 100 nodes), URL https://dev.lsstcorp.org/trac/wiki/db/Qserv/Testing