LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-404959

An Evaluation of Database Solutions to Spatial Object Association

V. S. Kumar, T. Kurc, J. Saltz, G. M. Abdulla, S. Kohn, C. Matarazzo

June 25, 2008

ACM SIG GIS
Irvine, CA, United States
November 5, 2008 through November 7, 2008

Disclaimer

An Evaluation of Database Solutions to Spatial Object Association*

Vijay S. Kumar, Tahsin Kurc, Joel Saltz
Department of Biomedical Informatics
The Ohio State University

Ghaleb Abdulla, Scott R. Kohn, Celeste Matarazzo
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

ABSTRACT

Object association is a common problem encountered in many applications. Spatial object association, also referred to as crossmatch of spatial datasets, is the problem of identifying and comparing objects in two datasets based on their positions in a common spatial coordinate system – one of the datasets may correspond to a catalog of objects observed over time in a multi-dimensional domain; the other dataset may consist of objects observed in a snapshot of the domain at a time point. The use of database management systems to the solve the object association problem provides portability across different platforms and also greater flexibility. Increasing dataset sizes in today's applications, however, have made object association a data/compute-intensive problem that requires targeted optimizations for efficient execution. In this work, we investigate how database-based crossmatch algorithms can be deployed on different database system architectures and evaluate the deployments to understand the impact of architectural choices on crossmatch performance and associated trade-offs. We investigate the execution of two crossmatch algorithms on 1) a parallel database system with active disk style processing capabilities, 2) a high-throughput network database (MySQL Cluster), and 3) shared-nothing databases with replication. We have conducted our study in the context of a large-scale astronomy application with real use-case scenarios.

1. INTRODUCTION

Object association, also known as cross-correlation or crossmatch, is the process of identifying and comparing objects or entities present in different datasets. Datasets in applications involving object association are often acquired from multiple observations of objects under varying experimental conditions and/or at different snapshots in time. In this work, we target the spatial object association problem (or the spatial crossmatch problem), in which objects are compared and matched based on their location in a multi-dimensional spatial domain. The object association problem is commonly encountered in many application domains. For example, a biomedical researcher may wish to match observations from multiple microscopic images to study the temporal evolution of cancerous cells within an organism. As another example, in astronomy, astronomers seek to crossmatch celestial objects captured at different wavelengths over time. In this case, objects observed by a telescope in a given time period may need to be compared and matched with objects in a catalog constructed from earlier observations.

The crossmatch problem can be defined as follows: Given two or more datasets of objects, the goal is to find for *each and every* object in one dataset all objects in the other datasets that lie within a certain "distance" of the object. Here, the notion of distance is defined based on a metric for the application under question. When the comparison between two objects is based on their positions in a common spatial domain, then the problem is referred to as the *spatial object association* or *spatial crossmatch* problem. The distance metric generally corresponds to the Euclidean distance or the angular separation between the objects in the spatial domain. In a GIS system, for instance, objects are spatially referenced to the earth. In such cases, crossmatch is based on their spatial proximity on earth.

Information about objects can be constructed from numerous surveys or observations using instruments, e.g., observation of objects in the sky by telescopes in an astronomy application. Each object has a unique identifier and a set of qualitative and quantitative features including its spatial coordinates. Object data are generally maintained in lists or catalogs. These catalogs are incrementally updated with newer observations of the same or different objects over time and hence the sizes of these catalogs may grow in time. Crossmatching of objects in moderately sized catalogs can be performed efficiently in reasonable time [11]. However, catalogs containing millions and billions of objects are increasingly becoming common in today's applications. The data and compute intensive nature of the problem presents a difficult challenge to the data management systems employed for such applications especially when the applications have real-time requirements. Crossmatch algorithms need to read from and update potentially multi-terabyte sized, disk-resident object catalogs. Hence, the performance of the crossmatch operation is dictated not only by the speed of evaluation

9*This research was supported in part by the National Science Foundation under Grants #CNS-0403342, #CNS-0426241, #ANI-0330612, #CNS-0203846, #ACI-0130437, #CNS-0615155, #CNS-0406386, and Ohio Board of Regents BRTTC #BRTT02-0003 and AGMT TECH-04049, the NIH U54 CA113001 and R01 LM009239 grants, and the NHLBI R24 HL085343 grant.

of the distance metric between pairs of objects, but also by the efficiency of data handling, i.e., the efficiency of data organization and storage, the performance of indexing, caching, and pre-fetching of data, and parallel processing of crossmatch operations. The choice of the underlying system for such data-intensive applications thus becomes an important decision that crucially impacts the application performance.

The catalog-based nature of datasets in applications involving object association operations naturally lends itself to the use of relational databases systems for data management. The ease of use and standardization of data definition and data manipulation languages make database systems a natural choice for storing large data. In this work, we investigate and evaluate different configurations of database systems on various architectures to study their impact on the performance of spatial object association (crossmatch). Clearly, a customized solution that is performance optimized for a specific architecture and the crossmatch operation will likely achieve better performance than a solution based on the use of a general database management system. However, there are some disadvantages in adopting such an approach; firstly, if similar problem requirements arise in a different domain, then the customized solution may not be directly applicable without making substantial modifications, whereas an existing database-oriented solution can modified more easily to suit the needs of the new application. Secondly, the vast amounts of data may be stored in publicly accessible data archives. Users may wish to modify application parameters or explore the data in many more ways than via specific operations like the crossmatch. In addition to providing generalized, flexible solutions to data selection and other exploration operations, database systems possess the important feature of portability across different architectures and computing platforms.

We focus on spatial crossmatch in the context of a large-scale astronomy application known as the Large Synoptic Survey Telescope (LSST) [8]. Our experiments are based on real use-case scenarios borrowed from the LSST. We argue that similar requirements will arise in other application domains including GIS. Hence, our findings can be applied in other application domains. The primary contribution of our work is that we provide an evaluation of and insights about how different parallel database management systems and architecture choices affect the performance of spatial crossmatch algorithms. We investigate the execution of two crossmatch algorithms known to be fast using 1) a parallel database management system with active disk style execution support for some types of database operations, 2) a database system designed for high-availability and high-throughput (MySQL Cluster), and 3) a distributed collection of database management systems with data replication.

2. APPLICATION DESCRIPTION

In large-scale astronomy studies, one or more high-resolution telescopes repeatedly capture images of the night sky over time. Object detection algorithms are used to detect and extract celestial objects from these images. These objects correspond primarily to light sources such as stars and galaxies. The object data and corresponding measurements are maintained in massive astronomy catalogs. A popular search query is the *points-near-a-point* query, in which for a given object, the catalogs have to be searched to determine all objects that lie within a certain angular separation or search radius from that particular object. The astronomy crossmatch problem belongs to the class of spatial crossmatch problems and is a generalization of the points-near-a-point query. The n-way astronomy cross-match query is as follows: "*Given n catalogs of objects, for each object belonging to one catalog, determine all potential matching objects from the remaining $n-1$ catalogs*", i.e. determine all objects that lie within a search radius of d arcseconds from that particular object.

The crossmatch query is useful in the following scenarios:

- When multiple collaborating astronomers capture the night sky using telescopes with different detection capabilities and wavelength settings from different geographic locations, their observations will be made under different conditions. Crossmatch helps to correlate their findings and provides a means to ratify their observations and accounting for unexpected phenomena.
- Due to the limited field-of-view (FOV) of the telescopes, the captured images generally correspond to smaller regions of the sky. When the same region of sky is observed multiple times, the data from these observations need to be "merged" accordingly so that up-to-date information for that region is available for further analysis and exploration. Crossmatch is an integral part of this merging operation. Snapshot data from every new observation is crossmatched against the historical catalog and the latter is updated with the results of the crossmatch.

The Large Synoptic Survey Telescope (LSST) [8] is a planned wide-field survey reflecting telescope that will photograph the available sky every three nights. The LSST has unprecedented data acquisition rates, courtesy of a 3.2 Gigapixel camera that captures an image every 15 seconds (12-15 terabytes a night). All acquired image data is archived and is expected to amount to 55 petabytes after 10 years [4]. The LSST catalogs are expected to contain around 50 billion objects at the end of the survey. One of the main scientific goals of the LSST is the ability to detect small objects and transient events that occur deep in the solar system.

Processing of image data occurs at a *Base Camp*, a computing center near sea level below the mountaintop telescope [4]. Data analysis pipelines have been constructed by astrophysicists for this purpose. The images are first processed through the *image processing pipeline* for correction and calibration. The *detection pipeline* extracts light sources from the images and generates detections by differencing with a template image (hence the detections are called Differential Image Analysis sources or DIASources). The *association pipeline* crossmatches these DIASources with existing objects in the historical catalog. That is, new detections from every snapshot are crossmatched against existing objects using a predetermined search radius. In database parlance, a crossmatch corresponds to a *spatial join* operation, i.e. a join between the catalog tables based on the spatial attributes of the objects. Here, we have a 2-way spatial crossmatch between the DIASource and object tables.

Each snapshot image corresponds to a single field-of-view (FOV). The real LSST estimates for an FOV size are about *10 sq.* degrees. The object density within an FOV is expected to be 10 million in the worst case and about 4 million in the average case. So, on an average, the association pipeline would involve a crossmatch of 40 thousand new detections against 4 million objects (hundred thousand against 10 million in the worst case). The biggest challenge

from a computational perspective is *real-time* transient alert generation [3], i.e. for any new unmatched detection, a decision must be made to trigger an "alert" in real-time so that astronomers the world over can be notified immediately thereby leading to closer monitoring of the transient object. Moreover, the object catalog needs to be updated on-the-fly to reflect newly detected objects. Hence, the performance requirements on the underlying computing system are extremely demanding.

The LSST use-cases and datasets possess chracteristics that can be exploited to improve performance: (1) The spatial extent of the sky in terms of *declination* and *right-ascension* coordinates is known in advance. So the spatial data structures and indexes can be constructed accordingly keeping these limits in mind. As objects are not expected to change position arbitrarily, reasonably static schemes can be employed to partition the sky. (2) The cadence of movement of the telescope as it scans the sky is also fairly regular. The LSST Online Control System will know at least 30 seconds in advance the coordinates of the next FOV. Thus, it is possible to prefetch the corresponding data for that FOV ahead of time. Also, it is assumed that the same FOV will not be visited more than once in some predefined interval. (3) The crossmatch of one FOV is independent of other fields of view. It is possible to pipeline the crossmatching operations for multiple fields of view, where the I/O operations for one FOV are coincident with the crossmatch computation for another FOV. The crossmatch process in the LSST is divided into three phases: (1) **prepare** phase where the object catalog is scanned and only the relevant data for the current FOV is staged into a separate table. (2) **compare and update** phase where the actual crossmatch is performed on the FOV data. (3) **postprocessing** phase that updates the object catalog based on findings from phase 2 and generates alerts if necessary. In section 3, we discuss some related efforts that address the crossmatch challenge.

3. RELATED WORK

Most efforts towards performance optimization of the spatial crossmatch problem for large datasets focus on reducing the computation cost (i.e., minimizing the number of object comparisons) and the disk I/O overheads. We classify these efforts into two broad categories: customized solutions and database-oriented solutions.

Customized solutions are those in which the crossmatch logic for the application is implemented outside of the database. These solutions are usually application-specific and involve the use of special data structures and specific optimizations for a given architecture. For the LSST application, Serge Monkewitz of the Infrared Processing and Analysis Center (IPAC) has developed a shared-memory based solution that performs real-time crossmatch on in-memory object data. As in most data-intensive applications, the LSST crossmatch operates only on small portions of the large dataset at a given time. That is, for a given FOV, it suffices to crossmatch detections against only those objects that lie within that FOV. Spatial indexes can be built on the data to speed up the process of selecting objects that lie within the bounding box of the FOV. Over the years, many spatial indexing schemes have been developed to support analysis of spatial data. Gaede and Gunther [6] provide a comprehensive survey of such schemes. Abdulla, a co-author on this paper, used variants of the R-tree index to support nearest-neighbor and other spatial queries in astronomy. The Hierarchical Triangular Mesh (HTM) spatial index by Kunszt [13] recursively divides the sky area into multiple spherical triangles and numbers them based on spatial proximity. Taylor [14] has proposed

Inputs:
Catalogs A, B
search radius θ
Algorithm:
1 $N \leftarrow$ size of A
2 $M \leftarrow$ size of B
3 **for** i = 1, N
3.1 **for** j = 1, M
3.1.1 **if** $d(A_i, B_j) < \theta$
3.1.2 **then**
3.1.2.1 Match
3.1.3 **endif**
3.2 **endfor**
4 **endfor**

Figure 1: Naive cross-match algorithm

$\mathcal{O}(N \log N)$ tile-based and multicone indexing approaches for the crossmatch. Papadomanolakis [10] developed an indexing algorithm known as Directed Local Search(DLS) for efficient query processing in unstructured tetrhedral meshes. Gray et al [7] propose a finer-grain "zones" indexing scheme that minimizes computational cost by reducing the search space for each object. While the use of indices improves data selection times, complex indexing schemes can slow down updates to the object catalog. Monkewitz has used data compression mechanisms to reduce disk I/O volume and facilitate efficient updates. Abdulla used incremental clustering techniques to update the catalog.

In contrast, crossmatch logic in database solutions is implemented in the database itself using the SQL language. Popular database systems support a basic set of native spatial indexing schemes such as the R-tree. Some of the more complex indexing schemes mentioned above may be implemented as stored procedures in the database. Power [11] has experimentally evaluated the performance of crossmatch algorithms on large catalogs using the MySQL and ORACLE database systems. Maria et al. [9] have implemented the crossmatch operation with the zones index on the Microsoft SQL Server system. Becla et al [4] offer an alternative view to data organization for the LSST. They describe the partitioned storage of the catalog in the form of smaller sub-tables, where each sub-table contains those objects that lie within a region or chunk of the sky. Sub-tables can be stored on or striped across multiple disks. No indices are maintained for the sub-tables. As a result, there is a large improvement in update performance with only an affordable increase in the data selection cost. Our work is similar to these earlier works in that we target database solutions to support spatial object association. We study the impact of architecture choices and parallel query processing techniques on crossmatch performance.

4. CROSSMATCH ALGORITHMS

In this section, we describe variants of the crossmatch algorithm that we evaluated on different configurations. The naive crossmatch algorithm, shown in Figure 1, takes each object in one table A and compares it against every object in another table B. Suppose table A has N objects and table B has M objects, then the cost of this all-to-all comparison approach increases as $\mathcal{O}(N \times M)$ and performs poorly for large catalogs.

4.1 Zones algorithm

In astronomy, the data space is a celestial sphere referenced using declination and right ascension coordinates that wraparound near the poles. Unlike adaptive mesh simulations and other spatial applications, the search radius is known in advance and is of the order of arcseconds [1]. The zones indexing algorithm proposed by Jim Gray et al. [7] bins the sphere data horizontally into non-overlapping "zones" or bands of some predefined height such that objects with similar declination values lie within the same zone. For example, if the zone height is chosen to be 1 arcminute, then there will be 10,800 zones. The characteristics of celestial objects have been well studied; for a given object, it suffices to look for matches within the same zone and within a small subset of "neighboring" zones. This way, the zones algorithm is able to reduce the search space and the computational requirements of the crossmatch operation[2]. An object within zone Z in one table is compared only against those objects in the other table that lie within neighboring zones of Z. Within each such zone, comparisons are made only against objects that are within a specific right-ascension range which is determined based on the spatial position of the zone. The crossmatch query expressed in SQL for the zones algorithm is shown in Figure 2. Here, A and B are the tables being crossmatched and θ is the search radius. *ZoneZone* is an intermediate precomputed table that maintains the neighbor zone information for each zone. ra and $decl$ are the right-ascension and declination coordinates for the object in the sky while *deltaRa* is the right-ascension range that needs to be searched within a zone. x, y and z are spatial coordiantes that allow for finer-grain distance-based comparison. This table is referred to as the intermediate zone neighbor table. The algorithm has been shown to work well for batch-oriented spatial queries like the crossmatch where the same operation is performed on multiple objects. An explanation of the query is beyond the scope of this paper. We refer the readers to the paper by Jim Gray et al. [7] for a detailed description of the algorithm.

```
SELECT A.objId, B.objId FROM A, B
INNER JOIN ZoneZone zz ON A.zoneId=zz.zoneId
INNER JOIN B ON zz.matchZoneId=B.zoneId
WHERE B.ra BETWEEN A.ra-zz.deltaRa AND A.ra+zz.deltaRa
AND B.decl BETWEEN A.decl-θ AND A.decl+θ
AND POW(A.x-B.x,2)+POW(A.y-B.y,2)+POW(A.z-B.z,2)< d_max
```

Figure 2: Zones algorithm

4.2 Optimized Zones Algorithm

Becla et al. [4] proposed an improved form of the zones algorithm that exploits specific features of the LSST. Their approach accounts for wraparound of the sky data near the poles. Moreover, their approach assumes that each zone has a maximum of three neighboring zones; the zone itself and the two "sandwiching" zones (i.e., $zone_{i-1}$ and $zone_{i+1}$ for a given $zone_i$) along the declination dimension. Through this assumption, they avoid the potentially costly join operation against the intermediate zone neighbor table in the zones algorithm. Instead, for a given zone in the primary table (table A in the zones algorithm), this algorithm creates a temporary "SecondaryZone" table on-demand containing all objects in the secondary table (table B in the zones algorithm) that are within the 3 zones of interest. Moreover, the "SecondaryZone" table contains only those attributes of each object that are relevant to the crossmatch, i.e. the spatial coordinates. Hence, the SecondaryZone table is smaller in size as compared to the intermediate zone neighbor table from the zones algorithm. A spatial join is then performed between the primary table and the SecondaryZone table thus resulting in a crossmatch of all objects that lie within the corresponding zone. This process is repeated for every zone in the primary table. The algorithm is shown in Figure 3.

Inputs:
Primary table A, Secondary table B
search radius θ
Algorithm:
1 $minZ \leftarrow$ minimum zone in A
2 $maxZ \leftarrow$ maximum zone in A
3 **for** $i = minZ, maxZ$
3.1 Determine neighboring zones of interest for zone i
3.2 Create temporary SecondaryZone table
3.3 INSERT INTO SecondaryZone
SELECT ra, decl, x, y, z, ObjId, zoneId
FROM B WHERE zoneId is a neighbor zone
3.4 SELECT A.ObjId, s.ObjId FROM A
INNER JOIN SecondaryZone AS s ON s.ra
BETWEEN right ascension limits
WHERE A.zoneId = i
AND s.decl BETWEEN A.decl - θ AND A.decl + θ
AND POW(A.x-s.x,2)+POW(A.y-s.y,2)
+POW(A.z-s.z,2) $< d_{max}$
3.5 Adjust SecondaryZone table (3-zone sliding window)
4 **endfor**

Figure 3: Optimized zones (*OptZones*) algorithm.

Compared to *OptZones*, the *Zones* algorithm uses a single complex query for the crossmatch involving joins between the primary and zone-neighbor tables as well as the secondary table. The joins are performed on the *zoneId* and *ra* attributes of an object. Appropriate indexes such as a composite B-tree index on these attributes should improve the join performance and overall query execution time. Good performance will transpire if the database system can use the proper indexes for the query. Joins are performed on a per-object basis against the intermediate zone neighbor table. Small zone sizes will result in a large zone neighbor table and increase the crossmatch time for this algorithm. The *OptZones* algorithm, on the other hand, removes the need for the large zone neighbor table. It breaks down the query execution into n joins between the primary table and smaller SecondaryZone tables created on-demand for each zone. Here, n is the number of zones present in a field of view (FOV). The *OptZones* algorithm basically seeks to emulate a sort-merge join operation between the two tables by joining them in an increasing order of the zoneIds. The ability to break down the query into subqueries allows for parallelization of the algorithm on suitable architectures. Our experiments evaluate the effect of different database configurations and architectures on these algorithms.

5. ARCHITECTURES AND DATABASE CONFIGURATIONS

[1] The projected search radius for the LSST is around 0.05 arcseconds.
[2] It is assumed that the height of a zone exceeds the search radius.

One of the challenges in the LSST is to determine the right kind of data management system to set up at the base camp to perform the nightly processing including the association pipeline. Database system configurations on a single processor system are unable to generate adequate performance to satisfy real-time constraints when dataset sizes increase. In this section, we present the system and database configurations we used in our experimental evaluation of the crossmatch algorithms. The base configuration is a stand-alone database system (the open-source MySQL version 5.1.22 was used) with a non-transactional storage engine (MyISAM in the case of MySQL). The server and client run on the same dual-processor machine.

5.1 Configuration 1: Parallel Database System with Active Disks

Our first configuration comprises of a database system, which runs on a massively-parallel backend and employs active disk style [2] hardware acceleration for some of the database operations. Active disk based systems push computation closer to the data, to the extent possible, as opposed to the conventional idea of staging disk-resident data into memory for processing. The use of active disks was initially proposed in order to offload bulk of the data processing to disk-resident processing elements [2]. Processing units are integrated with the disk drives to allow application-specific code to work on the data as it streams off the disk. The main motivations behind evaluating this configuration are to gauge the improvements brought about by parallel query execution (which crossmatch algorithm is more amenable to parallelization?) and by the use of active disk style processing (to what extent can disk I/O volumes be reduced by offloading processing to disk?). In our experimental evaluation, we used the Netezza Performance Server (NPS) [1], which is a commercial data warehousing appliance. The Netezza system stores data in a database that is distributed across multiple backend nodes. It also employs active disk style processing in that some of the database operations such as filtering are implemented in Field Programmable Gate Arrays (FPGAs) near disks. The system combines the use of the following features:

Asymmetric Massively Parallel Processing (AMPP): The system consists of one frontend and a large number of backend "snippet" processing units (SPUs). The frontend is responsible for query parsing, parallel query plan generation (i.e. each query is broken down into sub-queries or snippets that can be executed in parallel on the backend processors), and for combining the results obtained from multiple SPUs. Each SPU is made up of a low-power embedded PowerPC processor that has its own memory and disk (capable of delivering I/O bandwidths of upto 60 MB/sec). The aggregate bandwidth that one can obtain out of all the disks scales linearly with the number of SPUs in the system.

Active disks: Netezza implements a realization of active disk technology. Netezza provides a Field Programmable Gate Array (FPGA) processing unit attached to the disk drive on each SPU. The processing unit is programmed to perform simple filtering and projection operations on the data. These processors act as disk-controllers to filter the data as it streams from disk to the SPU's memory. In the current implementation, user-defined re-programming of the FPGAs is not allowed. Hence, the type of processing that can be pushed near disk is limited to the set of operations pre-programmed in the FPGAs.

High-speed interconnencts: The frontend and the SPUs are interconnected via a high-speed Gigabit Ethernet switch. By providing increased network bandwidth, the data exchange between the processing units is much faster and is not limited by the communications medium.

Hash-based data partitioning: Netezza uses a hash-based data partitioning technique to uniformly distribute the data in a table across all the SPUs in the system. The contents of each table in the database are striped row-wise and the records are then distributed across the SPU disks based on a hash of the contents of the distribution column. In general, a uniform hash-based distribution of a table's contents will lead to better load balance amongst the SPUs.

User-defined functions: Netezza provides support for user defined functions(UDFs), also known as *On SPU Functions (OSFs)*, which are application-specific functions executed in parallel by each SPU on its local data. UDFs are coded in a high-level language (C/C++) and translated into object code that can execute on each SPU.

Snippet processing: Each query is broken down into a set of sub-queries or snippets that can be executed in parallel on the SPUs such that most snippets require only data local to the SPU. In the case of joins and other complex queries, data needs to be transferred between SPUs over the high-speed interconnect. Such transfers occur in parallel without overloading the network.

Indices: Netezza does not support index construction on the data. In its place, Netezza provides zone-maps (not to be confused with the zones algorithm). That is, indices generally tell a database system what data to read, while zone-maps tell the Netezza system what *not* to read. Zone-maps are realized in a software module of Netezza known as the storage manager. Zone-maps exploit the natural ordering of data that exists in most tables. When a query arrives, the storage manager looks up the zone-map to tell the FPGAs what data extents not to read (row-based filtering). In addition, depending on the columns specified in the query, the FPGA drops the irrelevant columns of the table(column-based projection). Using these features, Netezza is able to provide appreciable performance even without index support.

We conducted our crossmatch experiments on the Netezza system set up at the Lawrence Livermore National Laboratory. We implemented the naive, *Zones*, and *OptZones* algorithms on the system. We used Netezza's SQL interface, *nzsql* which conforms to SQL standards and is a lighter version of PostgreSQL. As a result, we were able to use existing implementations of the different crossmatch algorithms on the Netezza system with minor modifications. The absence of certain features (mainly the lack of support for indexes and stored procedures) required re-writing of portions of the schema and the crossmatch queries for each algorithm.

5.2 Configuration 2: High-Throughput Network Databases

This configuration consists of a network database system designed to support multiple clients accessing the system over a network. High-throughput network databases can easily be set up to work on high-performance storage/compute clusters built from commodity off-the-shelf components. In our work, we employed MySQL Cluster as the high-throughput network database system.

MySQL Cluster [12] and its in-memory transactional storage en-

gine, NDB, present a virtual database to the end-user. The database is actually distributed across nodes in a shared-nothing architecture. The motivations behind evaluating such a configuration were two-fold: (1) MySQL Cluster uses the collective memory of all nodes to store data. The NDB storage engine maintains all tables and indexes in this memory pool. High-end storage clusters are generally built from memory-intensive machines. Given the rates at which memory capacity is increasing on today's machines and the rise in popularity of non-volatile flash memory, it may be feasible to store the entire object catalog in this memory pool. In addition, advances in high-speed interconnect technology such as Infiniband provide rapid access to remote memory. This way, the crossmatch can be executed entirely on in-memory data thereby leading to significant I/O savings. (2) MySQL Cluster is designed to be a high-throughput database system. Unlike the Netezza Performance Server that is designed to optimize execution of a complex query against large data, MySQL Cluster can support a large number of concurrent queries against the in-memory data (hence its designation as high-throughput network database system). Based on our observation that the crossmatch query in *OptZones* algorithm can be explicitly broken down into a large number of smaller sub-queries, the algorithm could benefit from a configuration like MySQL Cluster.

A MySQL cluster instance consists of a set of nodes in a shared-nothing environment, one of which acts as the manager node. The remaining nodes are either data nodes (the backend) or front-end nodes. Tables and indexes are partitioned and stored in memory on the data nodes. MySQL Cluster is also designed to be a high-availability database, where data can be replicated synchronously among the data nodes so that there is no single point of failure. The database system takes care of replica consistency. The in-memory data is also check-pointed to disk logs regularly in order to prevent data loss. Like Netezza, MySQL Cluster partitions data based on a hash of the distribution column(s) chosen by the user. However, MySQL Cluster does not have a parallel query engine. When a query is submitted, a front-end node parses the query and generates a query plan which is broadcast to all the data nodes. Index lookup and table scans occur in parallel on each data node, making MySQL Cluster efficient for point-queries and simple data lookup operations. For execution of joins and other complex queries, MySQL Cluster will not automatically transfer any data amongst the data nodes. The user will explicitly need to choose a data distribution strategy such that all joins operate on data local to each data-node.

For our experimental evaluation, we implemented the naive, *Zones*, and *OptZones* algorithms on a MySQL Cluster instance set up on an NSF-funded high-end memory cluster at the Ohio State University.

5.3 Configuration 3: Shared-nothing Databases with Replication

We designed a hybrid configuration using the MySQL database system, borrowing the features from the previous two configurations. This configuration runs over a cluster of data nodes in a shared-nothing architecture. Each node runs a server (the MyISAM storage engine was used) and operates only on its local data. The frontend consists of a master node that can break down queries into smaller sub-queries for execution on the data nodes. Data can be partitioned uniformly across the data nodes. Since this configuration does not support any parallel query execution, relevant data needs to be replicated. Like MySQL Cluster, indexes can be built on the data, database operations can be performed on in-memory tables and multiple queries can be issued concurrently from the master node. Like the Netezza, joins and other complex queries can be executed in parallel on the backend data nodes. In one extreme form of this configuration, we have no replication of data, i.e. the data is only partitioned across the backend nodes. Here, the query workload needs to be broadcast to all the nodes. At the other extreme, we have complete replication of data. Here, we can break down the query into sub-queries such that each data node takes on only a part of the workload.

We evaluated the *Zones* and *OptZones* algorithms on this configuration set up using the high-end memory cluster at the Ohio State University. Our master node executes outside of the database (written using C++). We maintain that this is not a customized solution for a specific application. The master node merely controls the partitioning and replication of data and schedules the execution of sub-queries among the backend nodes. Our implementation of this configuration uses DataCutter [5] for runtime support and parallel execution. DataCutter is a component-based middleware framework that uses the filter-stream programming model. In DataCutter, application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through a *stream* abstraction. In this work, we have used a version of DataCutter which employs the MVAPICH flavor of MPI as the message passing substrate. This enabled us to leverage the Infiniband support available on our cluster for high-speed data transfers.

6. PERFORMANCE EVALUATION

In this section, we describe dataset and query region characteristics and present experimental results for each configuration.

6.1 Dataset characteristics

As the LSST is not expected to start generating data until 2014, the USNO-B catalog is used to emulate the characteristics of the data catalogs that would be generated in the real LSST application. The USNO-B catalog is a public catalog generated by the National Optical Astronomy Observatory and contains over a billion objects. It is expected that the catalog would contain 23 billion objects by the time of the first public data release. "Skinny" object records use up 100 bytes while "fat" objects entries are about 1.1 kB. The actual LSST estimate for a fat object is around 1.7 kB. The crossmatch search radius used in all experiments is 3 arcseconds or 0.000833 sec. In our tests, we evaluated crossmatch performance only for skinny objects.

We use three different test FOV regions chosen by the astrophysicists to evaluate the crossmatch performance. These regions are characterized by differences in their object density (high, average, low). To simulate DIASources for a test FOV region, we use a perturbation function and apply it to the objects within that region[3]. The number of detections this function generates for an FOV is roughly one-hundredth the number of objects present within that region. Table 1 summarizes the characteristics for each of our test regions. The extreme-right column of the table is the number of matching entries that result from crossmatching the objects and DIASources regardless of the crossmatch algorithm used.

6.2 Stand-alone Database

[3]This function was developed by Serge Monkewitz of IPAC and his collaborators.

Test FOV Region	# Objects	# DIASources (approx.)	# resulting matches
High density	*3044468*	*30551*	*53938*
Average density	*373763*	*3709*	*4888*
Low density	*76073*	*764*	*942*

Table 1: Test Region Characteristics

In this base configuration, we use a MySQL client and server (version 5.1.22 with the MyISAM storage engine) on a dual-processor machine equipped with AMD Opteron-250 2.4 GHz processors, with 8 GB of DDR400 RAM. The machine has 2x250GB SATA Hard disks installed locally, joined into a 437GB RAID0 volume (with a RAID block size of 256KB). The maximum disk bandwidth available for sequential reads was 35 MB/sec and for sequential writes was 55 MB/sec. The object catalog data was partitioned into coarse chunks that were 1.75 degrees in height and 1.75 degrees in width. This coarse chunking strategy was proposed by Becla et al. [4] to minimize the prepare phase execution time. Object and detection data for a given FOV are extracted into in-memory tables for the crossmatch operation. Relevant indices are built on the in-memory tables to further improve performance.

Tables 2 and 3 respectively show the time taken to crossmatch the object table against the DIASource table and vice-versa for each algorithm and each test region. The order in which the tables are crossmatched can be modified without affecting the crossmatch result.

Test FOV Region	Naive algorithm	*Zones* algorithm	*OptZones* algorithm
High density	*11h 19m*	*11h 55m*	*17.05 s*
Average density	*10m 23s*	*10m 34s*	*1.9 s*
Low density	*25.57 s*	*27.15 s*	*0.4 s*

Table 2: Crossmatch time on MySQL (Object vs. DIASource)

Test FOV Region	Naive algorithm	*Zones* algorithm	*OptZones* algorithm
High density	*9h 21m*	*9h 58m*	*13.14 s*
Average density	*8m 26s*	*8m 53s*	*1.49 s*
Low density	*21.14 s*	*21.94 s*	*0.32 s*

Table 3: Crossmatch time on MySQL (DIASource vs. Object)

As expected, the naive algorithm performs poorly especially for the high-density region. One might expect the *Zones* and *OptZones* algorithm to perform much better as they reduce the search space. However, we observe that the performance of the *Zones* algorithm is no better than the naive algorithm. The performance figures suggest that an all-to-all comparison of objects and detections is occurring in the *Zones* algorithm for these queries and that the database system is not using the composite (*zoneId*, *ra*) index for the spatial join operations. The *OptZones* algorithm exhibits best performance. Here, the SecondaryZone table constructed on a per-zone basis is indexed only on *ra*.

6.3 Netezza Performance Server

A "half rack" Netezza [1] system at the Lawrence Livermore National Laboratory (LLNL) was used to evaluate configuration C2 described in section 5.1. The frontend of the system is connected via Gigabit Ethernet switch to 56 SPUs. Each SPU had 320 GB local disk with a read bandwidth of 60 MB/sec per SPU disk. Bulk upload mechanisms (nzload) were used to ingest the USNO-B data into the disks of these 56 SPUs. The data was distributed uniformly amongst the SPUs based on a hash of the object ID column. The selection of objects within a given FOV will occur in parallel on all SPUs. Unlike in the stand-alone database configuration where FOV objects are extracted into in-memory tables, here, they are extracted into another disk-based table that is also distributed across all the SPUs.

The prepare phase on the Netezza for the high, average and low density regions respectively took 87, 84 and 81 seconds. The time to insert data does not increase much with increasing object density. The crossmatch algorithms needed modifications to run on Netezza system. The *Zones* algorithm and *OptZones* algorithm had to run on non-indexed tables. The version of *nzsql* we used in this work did not have support for stored procedures. Hence, the **for** loop ranging from the minimum to the maximum zones in the *OptZones* algorithm had to be scripted explicitly outside of the database. The script issues a join query for each of the n zones in batch-form. Additional statements were added to the algorithm to remove transactional overheads that arise from batch execution of queries. We also developed a user-defined function (UDF) to determine the ra search-range for a given neighbor zone. This UDF is used in the modified *OptZones* algorithm and gets executed on each SPU.

Test FOV Region	Naive algorithm	*Zones* algorithm	*OptZones* algorithm
High density	*1950 s*	*51.15 s*	*210 s*
Average density	*25.62 s*	*2.8 s*	*96 s*
Low density	*2.1 s*	*0.69 s*	*67 s*

Table 4: Crossmatch time on Netezza (Object vs. DIASource)

Test FOV Region	Naive algorithm	*Zones* algorithm	*OptZones* algorithm
High density	*2739 s*	*52.47 s*	*280 s*
Average density	*36.79 s*	*2.85 s*	*132 s*
Low density	*1.9 s*	*0.7 s*	*118 s*

Table 5: Crossmatch time on Netezza (DIASource vs. Object)

Tables 4 and 5 show the performance numbers for crossmatch on the Netezza system. We observe that the *Zones* algorithm performs the best among all algorithms for every test region in spite of the fact that tables in Netezza are not indexed. This shows that the Netezza query engine can efficiently break down even complex join queries like crossmatch into appropriate snippets. The *OptZones* algorithm does not perform as well on the Netezza as it did on the stand-alone database configuration on account of two major reasons; firstly, the *OptZones* algorithm constructs and populates SecondaryZone tables on-demand for every zone in an FOV. These *ad hoc* tables will be created on disk as opposed to memory and distributed across all the SPUs in the Netezza system. The overhead of having to create disk-based tables on-demand leads to poor performance. Secondly, and to a lesser extent, stored procedures would have allowed us to prepare an SQL statement once and use it for multiple input values. The lack of support for stored procedures in the Netezza meant that each of the n queries issued from the external script were treated as independent queries and entailed query-parsing overheads each time.

It is interesting to note the effect of the order in which the tables are crossmatched on the crossmatch performance. As mentioned earlier, the DIASource table for an FOV contains roughly one-hundredth the number of elements present in the object table for that region. In the stand-alone configuration, it was better to crossmatch DIASources against Objects, whereas on the Netezza, it was better to crossmatch Objects against DIASources. We do not have an explanation for this observation apart from the fact that the behavior depends on the query plans generated by the query engine in each configuration.

The time taken to execute the prepare phase and the best crossmatch algorithm on the high-density region takes over two minutes on the Netezza system. We sought to avoid the prepare phase altogether by crossmatching DIASources directly against the large object catalog, a process that took 18 minutes. We observed that time to perform crossmatch increased exponentially as the size of the tables increased because larger tables imply greater data communication volumes among the SPUs. Thus, we partitioned the object catalog into a set of coarse chunks (represented as sub-tables in the database). Each sub-table is distributed across all the SPUs. The crossmatching of a pair of objects is independent of all other objects. So, when crossmatching tables $T1$ and $T2$, one can break down either of the tables, say $T2$, into n sub-tables $T2_1$, $T2_2$, ..., $T2_n$. Crossmatching $T1$ against each of these sub-tables and then simply merging the results has the same effect as crossmatching $T1$ directly against $T2$. Given an FOV and the corresponding DIASources, we determine the set of intersecting chunks and crossmatch the DIASources against the entire contents of each such chunk and merge the results. For the LSST, we uniformly partitioned the data space into disjoint coarse chunks that were 4.5 degrees wide and 4.5 degrees in height. The chunk tables are distributed amongst 48 SPUs of the Netezza system (8 of the SPUs were down at the time of testing). For this chunking strategy, the high-density FOV region intersected with 4 chunks. The results of the crossmatch against each of these four chunk tables is shown in Table 6. The last row in the table provides gives us the time it would take to directly crossmatch the DIASources against a table created from these 4 chunk tables using the prepare phase.

DIASource vs.	Crossmatch time	# matching entries
Chunk 1	*21 s*	*13196*
Chunk 2	*19.7 s*	*14424*
Chunk 3	*19.2 s*	*14140*
Chunk 4	*17 s*	*12178*
Total	*1m 17s*	*53938*
FOV object	*1m 11s*	*53938*

Table 6: Partitioned crossmatch time on Netezza (Chunks vs. DIASource)

The results tell us that crossmatch against a set of intersecting chunks performs only slightly worse as compared to the earlier 'prepare + crossmatch' strategy. This is understandable because there will be more false-positives to deal with in the former case. Importantly though, by using the intersecting chunks directly, we are avoiding the prepare phase altogether. To further reduce crossmatch time, we could simultaneously issue the crossmatch queries against each intersecting chunk. However, systems like Netezza are optimized for performance of complex queries on large data and not for throughput of execution of a large number of concurrent queries. We were limited here by the number of concurrent database sessions that Netezza can support.

6.4 MySQL Cluster

This configuration was tested using MySQL Cluster [12] (version 5.1.22, NDB storage engine) on an NSF-funded cluster at the Ohio State University consisting of 16 AMD Dual 250 Opteron nodes, each with 8 GB of memory. The nodes are interconnected by both an Infiniband and 1Gbps Ethernet network. Each node has 2x250GB SATA disks installed locally, joined into a 437GB RAID0 volume. The maximum disk b/w per node was around 35 MB/sec for sequential reads and 55 MB/sec for sequential writes. The MySQL Cluster configuration consisted of one manager node, upto 8 API nodes and 8 data nodes. The replication factor was disabled, i.e. there was just a single copy of the data partitioned uniformly among the data nodes.

Tables 7 and 8 show results for the crossmatch algorithms for the high and average density regions. For these experiments, we had a single API node issuing the crossmatch query.

Test FOV Region	Naive algorithm	*Zones* algorithm	*OptZones* algorithm
High density	*11h 22m*	*11h 58m*	*13m 34s*
Average density	*10m 16s*	*10m 57s*	*18s*

Table 7: Crossmatch time on MySQL Cluster (Object vs. DIASource)

Test FOV Region	Naive algorithm	*Zones* algorithm	*OptZones* algorithm
High density	*9h 27m*	*10h 2m*	*19m 40s*
Average density	*8m 37s*	*9m 15s*	*2m 30s*

Table 8: Crossmatch time on MySQL Cluster (DIASource vs. Object)

The results show that the naive and *Zones* algorithms perform no better on the MySQL Cluster than on the stand-alone MySQL configuration. Since the crossmatch is performed on memory-resident data in both configurations, we should not expect any improvement brought about via parallel I/O on the MySQL Cluster. Since execution of joins is not parallelized in MySQL Cluster, we presumed that relevant data is transferred to a single data node where the join is performed. However, the *OptZones* algorithm takes much longer on MySQL Cluster than on a stand-alone MySQL configuration. On further investigation, it was observed that joins and other complex queries in MySQL Cluster always get executed on a frontend API node. The *OptZones* algorithm breaks down the crossmatch query implicitly into a set of n joins, one per zone in the FOV. Hence for each join, data gets transferred over the network from the data nodes to a single API node where the join is executed. Moreover, there is very limited caching of data at the frontend, i.e. if we were to join table $T1$ against a hundred other tables, then, table $T1$ is transferred from the data nodes to the API node for each of the hundred join queries. This join mechanism explains the poor performance of the *OptZones* algorithm on this configuration.

The main advantage of the MySQL Cluster configuration is the ability to issue a large number of concurrent subqueries from multiple frontend API nodes. To exploit this feature, we broke down the query in the *Zones* algorithm explicitly into p sub-queries where p is the number of API nodes used to issue the sub-queries. In this way, each API node would submit a query responsible for (1/p)th of the overall query workload. Figure 4 shows the improvement in

query execution time as we increase the number of API nodes, i.e the number of concurrent sub-queries.

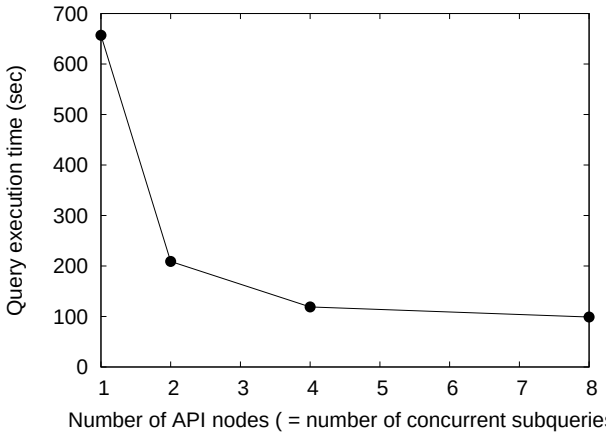

Figure 4: Improvement in crossmatch time with increase in API nodes on MySQL Cluster. (8 data nodes, average density test region

Although these numbers cannot be classified as real-time, they show that a configuration like MySQL Cluster can scale well as the crossmatch query is broken down into smaller and smaller subqueries. Improvements to the join mechansims are expected in future releases of MySQL Cluster.

6.5 Shared-nothing MySQL with Replication

This configuration was developed using the same hardware specified in Section 6.4. Each of the 16 nodes ran an independent MySQL server (version 5.1.22 with MyISAM storage engine) with table data stored on local memory/disk. An additional node served as a master node coordinated the query execution. This hybrid set up can be configured into independent groups of nodes based upon a replication factor. A set of n nodes can be divided into g groups of p nodes each. Data from the entire object catalog is distributed among the nodes in each group. Here, g is the replication factor because copies of data exist in each group, and there are g such groups. At one extreme, we may have an $n \times 1$ grouping, i.e. a single group consisting of n nodes. This corresponds to the case where the data is uniformly partitioned amongst the nodes without any replication (similar to the data distribution schemes in Netezza and MySQL Cluster). At the other extreme, we can have a $1 \times n$ grouping meaning we have n groups of one node each, where data is replicated across all the nodes, and a query can be executed indepenedently by any node/group. By modifying g and p for a given n, we can evaluate intermediary grouping schemes which help us control the data partitioning and replication more flexibly than in our earlier configurations.

We evaluated the *OptZones* algorithm which had to be modified to run in this configuration. The replication of the historical data amongst the nodes is done as an offline operation and is not part of our evaluation. Given a FOV, the master node first communicates the FOV to the worker nodes so that they can prefetch the historical objects within that FOV into an in-memory Object table (i.e. the prepare phase). The master node then sends the new DIASources for that FOV to the data nodes. Once the data nodes receive these detections, they perform the crossmatch in memory against only those objects that are stored locally on that node. In this confguration, our main goal is to be able to avoid the need for a distributed join operation. Since we are using a shared-nothing set of database instances, we need to structure the execution of the crossmatch queries such that each data node performs joins only on the data it locally owns.

The overall query execution time in this configuration includes the time to transfer the detections over the network and the time to execute the query. The amount of data transferred and the computational workload on each data node will depend upon the grouping strategy adopted. In the $n \times 1$ (only partitioning, no replication) case, every join operation potentially needs data from all nodes on account of the uniform partitioning of the Object table. Here, the master node needs to send the DIASources to all the data nodes. This could prove to be a bottleneck in the case where the number of DIASources is extremely large and comparable to the number of objects in the FOV, or in the case where our configuration has been deployed on a cluster with slow network connections between the nodes. This scheme would work well in the case where we have low-power processors connected via high speed communications medium. On the other hand, in the $1 \times n$ case, each join query can be executed by any one of the nodes. So, the master node divides the query workload equally amongst the data nodes and sends only (1/n)th the number of DIASources to each data node. In this case, we are reducing the volume of data communicated. But each data node will have to crossmatch its share of the DIASource against all objects in the FOV. This case would ideally suit clusters with very fast processors and slower networks. In the intermediate grouping strategies, the master node would need to send a subset of the DIASources to a group of data nodes.

FOV	Prepare time	DIASource time	Query transfer	Total time
High	*1.97 s*	*5.6 s*	*1.27 s*	*6.99 s*
Average	*0.87 s*	*5.05 s*	*0.24 s*	*5.39 s*
Low	*0.77 s*	*5.04 s*	*0.13 s*	*5.25 s*

Table 9: Execution time for 16x1 strategy (*OptZones* algorithm), Objects vs DIASources

Table 9 shows the execution times for each phase of the crossmatch in this configuration when we chose to simply partition the data without any replication. The "total" column is the sum of DIASource transfer time, the time to load the DIASources into memory on the data nodes and the query execution time and is measured as the execution time as perceived by the master.

Figure 5 shows the change in execution time of each component as we modify the grouping strategy. As we increase the replication factor, the data nodes spend more time on the prepare phase(not shown) and more time on the query execution. However, the transfer time decreases, although only marginally for our test case.

7. SUMMARY AND CONCLUSIONS

In this work, we have explored and evaluated database-based solutions to spatial object association or crossmatch, an important spatial data analysis operation that finds use in diverse application domains ranging from astronomy to GIS. We have investigated two variations of spatial crossmatch algorithms, the *Zones* [7] and *Optimized Zones* (*OptZones*) [4]. These two algorithms implement different query styles and optimizations. We have performed an experimental evaluation of the two algorithms on 1) a

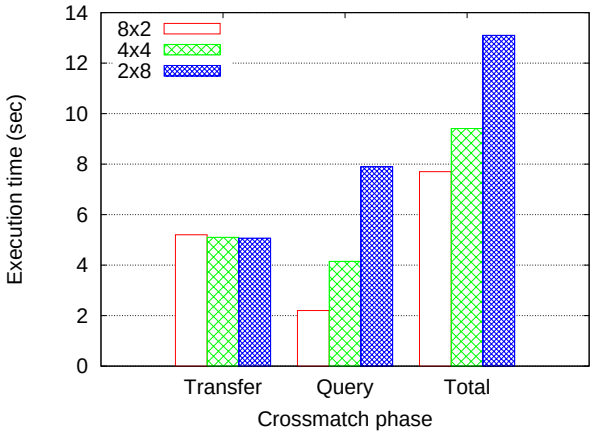

Figure 5: Varying replication factors and query partitioning mechanisms (*OptZones* algorithm, high-density test region, DIASources vs Objects)

parallel database system with active disk style support for certain database operations, 2) a database designed for high-availability and high-throughput, and 3) shared-nothing configuration of multiple instances of sequential databases with data replication. We tuned the algorithms to execute on these different database systems. All of our experimental results were based on real use-case scenarios put together by astrophysicists from the LSST.

Our experimental evaluation shows that

- The Zones algorithm performs better than the OptZones algorithm on the first database system configuration, because 1) the *OptZones* algorithm constructs and populates SecondaryZone tables on-demand for every zone in an FOV. These tables are created on disk as opposed to memory and distributed across all the SPUs in the Netezza system. The overhead of having to create disk-based tables on-demand leads to poor performance; 2) The lack of support for stored procedures in the system resulted in overheads due to external scripts being treated as independent queries.

- On the second database system configuration, the algorithms do not perform as well as they do on the first configuration. This is because the second configuration does not execute a query in parallel. However, the performance of the algorithms can be improved by partitioning a query into a set of smaller queries and executing these queries as a batch, to take advantage of the high-throughput oriented design of the second configuration.

- The third configuration enables different partitioning and replication of the catalog tables across the instances of the database management system in the system. The performance results obtained from the *OptZones* algorithm on this configuration indicates that the query execution time increases as the amount of replication increases. This is because while replication reduces the amount of DIASources entries broadcast to multiple nodes, but increases the time for the prepare and query phases, since each node has to deal with a larger portion of the catalog table (the Objects table). In our case, the size of the DIASources was relatively small. It can be expected that if DIASources is very large, then a configuration with replication of portions of the catalog table could be more efficient, since replication will reduce the amount of DIASource entries broadcast.

8. ACKNOWLEDGEMENTS

The authors wish to thank the late Marcus Miller for his help in configuring the Netezza systems at Lawrence Livermore National Laboratory (LLNL) and Serge Monkewitz of the Infrared Processing and Analysis Center(IPAC) for providing access to relevant algorithms and datasets. The authors also wish to thank Sergei Nikolaev and Don Dossa of LLNL for their help in understanding LSST application details.

Prepared by LLNL under Contract DE-AC52-07NA27344.

9. REFERENCES

[1] The netezza fast engines framework: A powerful framework for high-performance analytics. Netezza Technical White Paper, 2007.

[2] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, 1998.

[3] J. Becla, A. Hanushevsky, S. Nikolaev, G. Abdulla, A. S. Szalay, M. A. Nieto-Santisteban, A. Thakar, and J. Gray. Designing a multi-petabyte database for lsst. *CoRR*, abs/cs/0604112, 2006.

[4] J. Becla, K.-T. Lim, S. Monkewitz, M. Nieto-Santisteban, and A. Thakar. Organizing the extremely large lsst database for real-time astronomical processing. 17th Annual Astronomical Data Analysis Software and Systems Conference (ADASS 2007), London, England, 23-26 Sep 2007.

[5] M. Beynon, R. Ferreira, T. M. Kurc, A. Sussman, and J. H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, pages 119–134, 2000.

[6] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[7] J. Gray, M. A. Nieto-Santisteban, and A. S. Szalay. The zones algorithm for finding points-near-a-point or cross-matching spatial datasets. *CoRR*, abs/cs/0701171, 2007.

[8] Z. Ivezic, J. A. Tyson, R. Allsman, J. Andrew, R. Angel, and f. t. L. Collaboration. Lsst: from science drivers to reference design and anticipated data products. 2008.

[9] M. A. Nieto-Santisteban, A. R. Thakar, and A. S. Szalay. Cross-matching of very large datasets. In *National Science and Technology Council(NSTC) NASA Conference*, 2007.

[10] S. Papadomanolakis, A. Ailamaki, J. C. López, T. Tu, D. R. O'Hallaron, and G. Heber. Efficient query processing on unstructured tetrahedral meshes. In *SIGMOD Conference*, pages 551–562, 2006.

[11] R. A. Power. Large Catalogue Query Performance in Relational Databases. *Publications of the Astronomical Society of Australia*, 24:13–20, May 2007.

[12] M. RonstrÃűm and L. Thalmann. Mysql cluster architecture overview. MySQL Cluster Technical White Paper, 2004.

[13] A. S. Szalay, J. Gray, G. Fekete, P. Z. Kunszt, P. Kukol, and A. Thakar. Indexing the sphere with the hierarchical triangular mesh. *CoRR*, abs/cs/0701164, 2007. informal publication.

[14] M. Taylor. Crossmatching developments. *DS3 Report, VOTech Stage 6 Planning Meeting*, 2007.