Automatic Identification and Classification of Palomar Transient Factory Astrophysical Objects in GLADE

Weijie Zhao

University of California Merced, USA
E-mail: wzhao23@ucmerced.edu

Florin Rusu

University of California Merced, USA
E-mail: frusu@ucmerced.edu

John K. Wu

Lawrence Berkeley National Laboratory, USA
E-mail: kwu@lbl.gov

Peter Nugent

Lawrence Berkeley National Laboratory, USA
E-mail: penugent@lbl.gov

Abstract: Palomar Transient Factory is a comprehensive detection system for the identification and classification of transient astrophysical objects. The central piece in the identification pipeline is represented by an automated classifier that distinguishes between real and bogus objects with high accuracy. The classifier consists of two components—real-time and offline. Response time is the critical characteristic of the real-time component, while accuracy is representative for the offline in-depth analysis. In this paper, we make two significant contributions. First, we present an experimental study that evaluates a novel implementation of the real-time classifier in GLADE—a parallel data processing system that combines the efficiency of a database with the extensibility of Map-Reduce. We show how each stage in the classifier – candidate identification, pruning, and contextual realbogus – maps optimally into GLADE tasks by taking advantage of the unique features of the system—range-based data partitioning, columnar storage, multi-query execution, and in-database support for complex aggregate computation. The result is an efficient classifier implementation capable to process a new set of acquired images in a matter of minutes even on a low-end server. For comparison, an optimized PostgreSQL implementation of the classifier takes hours on the same machine. Second, we introduce a novel parallel similarity join algorithm for advanced transient classification. This algorithm operates offline and considers the entire candidate dataset consisting of all the objects extracted over the lifetime of the Palomar Transient Factory survey. We implement the similarity join algorithm in GLADE and execute it on a massive supercomputer with more than 3000 threads. We achieve more than 3 orders of magnitude improvement over the optimized PostgreSQL solution.

Keywords: Parallel Databases; Multi-Query Processing; Scientific Data Analysis; Similarity Join; Astronomical Surveys; Transient Identification

1 Introduction

The Palomar Transient Factory (PTF) project [Palomar Transient Factory (2016); Law et al. (2009)] aims to identify and automatically classify transient astrophysical objects such as variable stars and supernovae in real-time. As a secondary objective, a catalog containing the identified transients and other celestial objects is constructed for subsequent querying and analysis. PTF is a comprehensive transient detection system including a wide-field survey camera, an automated real-time data reduction pipeline, a dedicated photometric follow-up telescope, and a full archive of all detected sources (Figure 1 [Law et al. (2009)]). The computational system supporting the project consists of two separate processing pipelines [Law et al. (2009)] fed with the images taken by the camera. Between 2000 and 4000 high-resolution (2048 × 4096 pixels) images are taken each night and fed into the two pipelines through high-speed communication links. The total amount of raw data varies between 60 and 100 GB per night. The near-real-time transient detection pipeline [Bloom et al. (2011)] has the goal of identifying and classifying transient objects within 30-45 minutes of images being taken. Observation of potential transients by a network of follow-up telescopes is triggered immediately after detection in order to confirm their existence. The main objective of the time-consuming archival pipeline [Grillmair et al. (2010)] is to create a comprehensive catalog of high-quality images and celestial objects that can be queried using a variety of criteria. It is executed on the entire set of images acquired during one night in order to achieve high accuracy. The execution of the archival pipeline typically takes 4-5 hours [Grillmair et al. (2010)], but it can extend to several days in some cases [Law et al. (2009)].

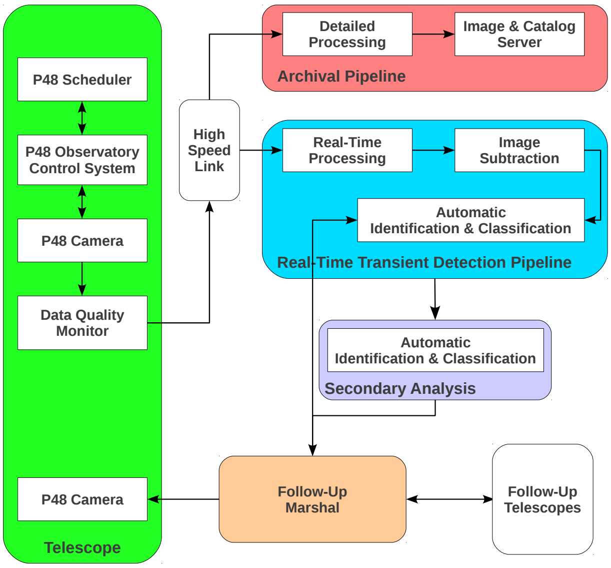

Figure 1: PTF data flow [Law et al. (2009)].

Problem formulation. The problem we address in this paper is the identification of real transient candidates in the detection pipeline. Specifically, we focus on the classification phase of the real-or-bogus classifier. The goal of this classifier is to identify real transients with high accuracy. The input consists of a set of candidates extracted during image subtraction and a trained random forest classifier. In the output, the candidates are given scores, i.e., the realbogus score, quantifying the probability of them being real. Only the candidates with realbogus score higher than a threshold and satisfying a set of additional constraints are considered real.

Contributions. We target two objectives corresponding to transient classification. The first objective is to design and implement a real-time classifier capable to keep-up with the continuously increasing size of the PTF repository. Our motivation is the incapacity of the existing PostgreSQL solution to identify transient candidates accurately due to the larger data volumes it has to handle. We present a novel implementation for the real-or-bogus classification in GLADE [Cheng et al. (2012)]—a parallel multi-query processing system targeted specifically at analytical workloads. We show how each stage in the classification process is natively supported in GLADE – this is not true for the existing PostgreSQL solution – and prove with experimental results the effect on query execution performance. Since the GLADE implementation reduces the time to investigate a set of candidates to minutes – from hours in PostgreSQL – this allows for considerably more candidates to be thoroughly evaluated, thus increasing the likelihood to find many transients that are otherwise missed by the current solution. These results have been presented before in a shortened version of the paper [Rusu et al. (2014)].

The second objective is to design a holistic parallel algorithm that considers the entire candidate dataset when classifying potential candidates. The motivation for such an algorithm is the extensive pruning applied by the real-time classifier in order to keep-up with the strict time constraints and the limited time horizon used in classification. Our solution is to add a secondary data analysis component to the PTF architecture—executed during the telescope offtime. In this component, we compute all the candidate pairs that are within a specified distance of each other across the entire PTF survey and apply a series of statistical functions to the resulting dataset. This corresponds to a similarity join query across the candidates—problem known to be extremely difficult. Our contribution is a new parallel similarity join algorithm that builds a dynamic index incrementally and allows for multiple pairs to be processed in batches. We implement the similarity join algorithm in GLADE and execute it on a massive supercomputer with more than 3000 threads, achieving more than 3 orders of magnitude improvement over the optimized PostgreSQL solution. These contributions are completely new with respect to the previous conference version of the paper [Rusu et al. (2014)]. The scientific benefit of secondary analysis is that it allows for the retrospective evaluation of potentially mis-classified or undetected transients.

Outline. Section 2 presents automatic transient identification and real-or-bogus classification in detail. It also discusses the PTF solution deployed in production and the problems it has. Section 3 introduces the novel parallel similarity join algorithm designed for secondary analysis. GLADE is introduced in Section 4. The implementation of the real-or-bogus classifier in GLADE and the results of the experimental evaluation are given in Section 5 which also contains a comparison with two PostgreSQL solutions. We conclude the paper in Section 6.

2 Automatic Transient Identification and Classification

In this section, we present the details of the *automatic detection and classification stage* in the real-time transient detection pipeline (Figure 1 [Law et al. (2009)]). The input to this stage is represented by the transient candidates extracted during the image subtraction stage. There are in the order of 10^5 such candidates extracted every 45 minutes. Two questions have to be answered for every candidate:

1. Is the candidate real?
2. If real, what is the transient type of the candidate?

Both these questions are answered using automated machine learning classification techniques, i.e., random forest classifiers [Bloom et al. (2011)] in this case, that require human intervention only in the follow-up stage. This is necessary considering the number of candidates – 1 to 1.5 million – extracted every night. Since the focus of this work is identifying real candidates – the first question – we present the details and the existing solution in the following. A description of the type classifier can be found elsewhere [Bloom et al. (2011)].

2.1 Real-or-Bogus Classification

Any machine learning method consists of multiple phases. First, a series of features have to be defined for the input data. These are used as parameters for the classifier. The features used by the real-or-bogus classifier are extracted during image subtraction and stored in the candidate database. There are 28 features used by the classifier. Second, a training dataset containing labeled examples is used to compute the parameters of the classifier—the training phase. The training dataset consists of 574 candidates manually labeled with the realbogus score by multiple human scanners. At last, the trained classifier is presented with unlabeled examples and the class has to be determined—the classification phase. The output consists of 5 classes, i.e., {bogus, suspect, unclear, maybe, realish}. The probability of a candidate being in each of these classes is returned by the random forest classifier—probabilistic model. The score, i.e., realbogus score, for a candidate is computed as a weighted average of the class probabilities. The final realbogus score takes into account additional information, i.e., the scores of neighboring candidates from the same subtraction. Moreover, a high-scoring candidate is deemed real if and only if it appears in at least two subtractions within 6 days. All the candidates identified as real at the end of the classification phase – 30 to 150 out of 10^5 – are flagged for immediate follow-up and sent to the type identification classifier. We present the exact details of the online classification phase in the following since training is a one-time offline process.

Algorithm 1 *Real-or-Bogus Classification*

Input: new subtraction set (subtraction) with corresponding candidate set (candidate) and their probabilities (rb_classifier) computed by the random forest classifier

Output: a subset of real candidates (real)

```
real ← IdentifyCandidates(subtraction,
    candidate, rb_classifier)
for all r ∈ real do
  if SingleAppearance(r, subtraction,
      candidate, rb_classifier) then
    real ← real - r
  end if
end for
for all r ∈ real do
  ctxt_score ← CtxtRealBogus(r,
      subtraction, candidate,
      rb_classifier)
  if ctxt_score < threshold then
    real ← real - r
  end if
end for
```

Identify candidates. The initial realbogus score – the score returned by the random forest classifier – corresponding to a candidate is computed during image subtraction. It is stored together with other candidate data in the subtraction and candidate database. Candidate identification requires a simple query that returns all the candidates with high realbogus score extracted from subtractions computed during a specified time interval.

Prune single appearance candidates. In order to increase the probability that a high-scoring candidate is indeed real, a candidate identified in the first query has to satisfy an additional condition. The candidate has to appear at a close spatial position in other subtractions close in time to its originating subtraction. Independent of its original realbogus score, a single appearance candidate is pruned away. Since the area and time interval are dependent on the candidate, pruning requires a separate query with different space and time bounds for every candidate. The larger the number of candidates identified in the first query, the more queries have to be executed for pruning.

Compute contextual realbogus score. The final realbogus score of a candidate takes into consideration the score returned by the random forest classifier for the closest k other candidates extracted from the same subtraction. This is called the contextual realbogus score since it considers the candidate in the larger context of all the extracted objects close in space. It is based on this score that the final classification decision is made. Computing the contextual realbogus score is a rather complicated process that involves a nearest-neighbor query followed by a complex aggregate computation. Unlike pruning, the nearest-neighbors are computed only along the spatial dimension, i.e., in the same subtraction. Nonetheless, a separate query has to be executed for every candidate that survives pruning—a considerable problem when the number of candidates is large. Algorithm 1 summarizes formally the stages of the real-or-bogus classification.

2.2 Secondary Analysis

As mentioned in the introduction, the goal of the secondary analysis stage is accuracy rather than time efficiency. This stage is executed during daylight, when the telescope does not acquire new images. The algorithm executed during secondary analysis eliminates the pruning steps from Algorithm 1 and extends IdentifyCandidates to operate over the complete set of candidates ever extracted. Thus, the observation horizon is considerably larger than the subtraction – or a small set around the subtraction – from which the candidate is extracted. Essentially, the time dimension is eliminated from identification which is done exclusively based on the spatial coordinates of the candidates. Abstractly, this corresponds to a similarity join query over the spatial coordinates.

2.3 Existing Solution

The current real-time solution implemented in the PTF pipeline is a standard Python application with a PostgreSQL database backend. The database contains 3 tables – `subtraction`, `candidate`, and `rb_classifier` – storing subtractions, candidates, and the scores returned by the random forest classifier. `subtraction` and `candidate` are wide tables having 51 and 46 columns, respectively. Many of the columns are never used in queries. The number of rows in these tables increases continuously as more observations are taken daily. `candidate` and `rb_classifier` already contain a few billion tuples each. Data corresponding to a new set of images are added to the tables during the image subtraction stage. Transient identification and classification execute as database queries. Possible candidates in a given time window are identified with a complex query over the 3 tables. For each such candidate, a time and space nearest-neighbor query is executed from the application to find additional appearances of the candidate. These queries are executed iteratively. For the remaining candidates, another spatial nearest-neighbor query is executed to find close candidates in the same subtraction. The contextual realbogus score is computed in the application by combining the score of the candidate with those of its neighbors. This is another iterative process that goes over the non-pruned candidates one at a time. The secondary analysis stage is not part of the current PTF production pipeline due to performance considerations. The algorithm we introduce in this paper will allow for this to happen in the future.

Problems with the existing solution. The existing approach suffers from a series of problems that make real-time candidate identification and classification in the limited time interval between two subtraction sets – approximately 45 minutes [Bloom et al. (2011)] – difficult. Experimental results over a relatively small snapshot of the database from the early stages of the project confirm this problem (Section 6). As the size of the repository increases with the acquisition of new images, the situation will become only worse. As a result, many of the transients are missed simply because there is not enough time to investigate them.

The fundamental limitation is raised by the need to evaluate each candidate sequentially even though the same query template is used across all the candidates. Essentially, two passes over the candidates extracted based on the realbogus score are required to take a decision. And in each pass, a complicated nearest-neighbor query is executed for each candidate. Since the time taken to process one query over the increasingly larger `candidate` table grows continuously with the size of the table, the number of candidates that can be inspected between two subtractions decreases.

The PostgreSQL row-based storage format affects query performance negatively considering the width of `subtraction` and `candidate` tables and the number of attributes used in the query—a small fraction out of the overall number of attributes. While query execution speed can be improved with appropriate indexes, this results in data ingestion time increase due to index maintenance, thus limiting the time available for querying. Essentially, indexing moves the bottleneck from querying to data ingestion. With the increase in repository size, index maintenance under data ingestion only becomes worse. Data transfer between the database and the application is another limitation that is a direct consequence of the large number of queries that have to be executed. The reason for this is the lack of support for complex computations inside the database. While user-defined functions (UDF) and user-defined aggregates (UDA) provide extensibility to in-database complex computations, the exclusive SQL invocation limits their applicability. As a result, these complex computations are executed in the application in the current PTF solution.

The implementation of the secondary analysis in PostgreSQL uses the `q3c` package [Koposov and Bartunov (2006)]. The initial results are not encouraging at all. It is impossible to execute the complete similarity join query across the entire dataset. Moreover, it takes 20 ms to compute the spatial neighbors of a single candidate—this corresponds to more than 150 days overall.

3 Secondary Analysis

In this section, we discuss the problem of secondary analysis in detail and present a fast and scalable solution. After real-time pipeline processing, each object is stored as a tuple (idx, ra, dec), where idx is the index of the object and ra and dec, respectively, correspond to the object position in the equatorial coordinate system. This stage takes a list of objects as input, and finds all the neighbors within threshold c for each object—distance-based similarity join query. c is a manually-set small constant, e.g., $1/3600°$.

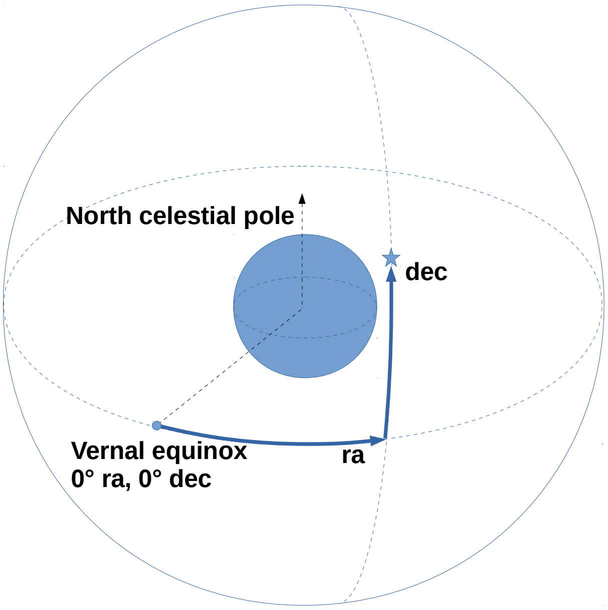

Figure 2: Equatorial coordinate system.

Equatorial coordinate system. The spherical coordinates of a star are commonly expressed as ra and dec in the equatorial coordinate system. Figure 2 depicts how to find the corresponding coordinate of a celestial object, where dec is the angular distance perpendicular to the Earth's Equator and ra measures the angular distance from the vernal equinox Eastward to the object's projection on the Equator. The vernal equinox is the origin point of the coordinate system which corresponds to $(0°ra, 0°dec)$. The ranges of ra and dec are $[0, 360)$ and $[-90, 90]$, respectively.

The computation of this stage can be expressed as the following self-join SQL query:

```
SELECT a.idx, b.idx
FROM obj_list AS a, obj_list AS b
WHERE dist(a.ra,a.dec,b.ra,b.dec) <= c
```

where *dist* is a user-defined function to compute the distance between two objects in equatorial coordinate system. Exact distance computation in the equatorial coordinate system requires multiple trigonometric function calls. However, the distance can be approximated as in Eq. (1) when c is small:

$$dist(ra_1, dec_1, ra_2, dec_2) = \sqrt{(ra_1 - ra_2)^2 \cdot cos^2(dec_1) + (dec_1 - dec_2)^2} \quad (1)$$

Since each object has to be joined with all the other objects detected before, including those objects detected in previous months or even years, the list of objects contains hundreds of millions of pairs. Executing the self-join query in a naive way requires producing more than 10^{16} pairs, which is impractical.

The standard solutions [Das Sarma et al. (2014)] to execute the similarity self-join query in acceptable time require the construction of spatial data partitioning indexes, e.g., quad-tree, kd-tree, that store the spatial information of the data. Then, they iterate over the objects and find their neighbors. The index accelerates the retrieving process by avoiding checking all other objects. However, the index-based methods execute the query for each object independently. The relationship among queries is ignored so that unnecessary repeated index accesses make the overall process inefficient. We propose a *novel method to dynamically build a lightweight index on-the-fly and process the queries for all the objects in a batch*. Thus, our solution eliminates the excessive cost of frequent index lookups for each object individually.

Data repartitioning. The original data are sorted by their *idx* value. *idx* does not correspond to the spatial position (ra, dec). It is necessary to ensure the neighbors of each object are stored close enough, so that the original data can be repartitioned with small overlapping. Unlike the Euclidean coordinate system, we observe that ra and dec are asymmetric in the distance function in Eq. (1). When $cos(dec)$ is close to 0, ra contributes almost nothing to the distance. On the other hand, when $|dec_1 - dec_2| > c$, *dist* is larger than c. Based on this observation, we sort the original data on ra in ascending order.

Dynamic data allocation. Each worker thread is dynamically assigned a corresponding data partition. Shared parallel file systems are commonly used in supercomputer centers, e.g., Lustre. Each partition is represented as a pair *(start,size)*, where *start* and *size* are the starting position offset in the file and the size of the partition, respectively. These offsets and sizes can be determined dynamically at runtime, after the number of worker threads is set.

Algorithm 2 *Partition Processing*

Input: A partition P contains n objects. Objects are sorted in the ascending order of dec.
Output: $neighbor_i$, neighbors of the i^{th} object

```
 1: Initialize an empty balanced tree $T$ whose key is $ra$
 2: for each $P_i \in P$ do
 3:   $neighbor_i = \emptyset$
 4:   $\delta = \frac{c}{cos(P_i.dec)}$
 5:   for each $P_j \in T.range(P_i.ra - \delta, P_i.ra + \delta)$ do
 6:     if $P_j.dec < P_i.dec - c$ then
 7:       $T.erase(P_j)$
 8:     else if $dist(P_i.ra, P_i.dec, P_j.ra, P_j.dec) \leq c$ then
 9:       $neighbor_i = neighbor_i \cup P_j$
10:       $neighbor_j = neighbor_j \cup P_i$
11:     end if
12:   end for
13:   $T.insert(P_i)$
14: end for
```

Query processing. We process each object in a partition iteratively and answer the query efficiently by maintaining a balanced tree T. Algorithm 2 depicts the computation details. $T.range(lower\_bound, upper\_bound)$ extracts objects whose key is located in $[lower\_bound, upper\_bound]$. $T.insert/erase(P_i)$ inserts/erases object P_i into/from the tree. According to the distance function in Eq. (1), only objects whose ra is located in $[P_i.ra - \delta, P_i.ra + \delta]$ and dec is located in $[P_i.dec - c, P_i.dec + c]$ are possible to be the neighbor of P_i. These objects can be found efficiently by querying T. Lines 6 and 7 check whether an object is impossible to be the neighbor of the following unprocessed objects in the list and erase it from T in order to make queries on T faster and more memory-efficient.

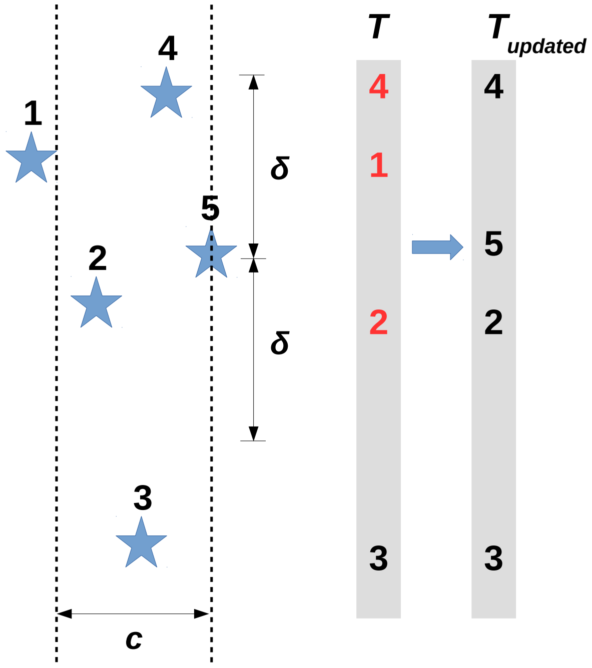

Figure 3: An example for partition processing.

An example for Algorithm 2 is illustrated in Figure 3. All the objects are sorted in the ascending order of dec. The example shows a snapshot during the entire partition processing. Consider we are processing the 5^{th} object, P_5. The previous 4 objects are stored in T, ordered by their ra. First, we compute the value of δ ($\delta = c/P_5.dec$). By searching the lower bound of $P_5.dec - \delta$ and the upper bound of $P_5.dec + \delta$ on T, we obtain a list of potential neighbors of P_5, $\{P_4, P_1, P_2\}$. Then we iteratively go over the list, and check the distance of each object in the list to P_5. Specifically, when we process an object whose dec is less than $P_5 - c$, we immediately delete it from T without checking the distance. When we process P_1, $P_1.dec < P_5.dec - c$, thus it can never be the neighbor of any subsequent object. Therefore, P_1 is deleted from T. After these steps, we insert P_5 into T.

Merging. The borders of partitions are handled in the merging stage. Consider two adjacent partitions a and b. Objects in the tail of a are possible to be the neighbors of objects in the head of b. Algorithm 3, a modification from Algorithm 2, shows the process to merge the border results. *extract_tail/head*(P, c) extracts objects in the tail/head of P whose dec are no smaller/greater than $min_{dec}(P) - c$ / $max_{dec}(P) + c$. $min/max_{dec}(P)$ is the dec of the first/last object, since all objects are sorted by dec. The algorithm inserts objects from one partition into the balanced tree and then iteratively visits the objects from the other partition without inserting them into T. Therefore, only neighbors across partitions are computed by this process. Moreover, we guarantee that $max(dec) - min(dec) > c$ for each partition in order to avoid merging non-adjacent partitions.

Algorithm 3 *Border Merging*

```
Input: Two adjacent partitions $P$ and $P'$
Output: $neighbor_i$, neighbors of the $i^{th}$ object
Initialize an empty balanced tree $T$ whose key is $ra$
$P_{tail} = extract\_tail(P, c)$
$P'_{head} = extract\_head(P', c)$
for each $P_i \in P_{tail}$ do
    $T.insert(P_i)$
end for
for each $P_i \in P'_{head}$ do
    $neighbor_i = \emptyset$
    $\delta = \frac{c}{cos(P_i.dec)}$
    for each $P_j \in T.range(P_i.ra - \delta, P_i.ra + \delta)$ do
        if $P_j.dec < P_i.dec - c$ then
            $T.erase(P_j)$
        else if $dist(P_i.ra, P_i.dec, P_j.ra, P_j.dec) \leq c$ then
            $neighbor_i = neighbor_i \cup P_j$
            $neighbor_j = neighbor_j \cup P_i$
        end if
    end for
end for
```

Complexity analysis. The time complexity of Algorithm 2 is $\mathcal{O}(n \times \eta)$, where n is the number of objects, and η is the average number of neighbors. The algorithm can be divided into 3 stages: inserting/deleting objects, querying ranges, and traversing the range. Each object is inserted/deleted into/from T only once. Thus, the time complexity of this stage is $\mathcal{O}(n \log n)$. Each range query on T takes $\mathcal{O}(\log n)$ time which is a native function of balanced trees. In addition, range queries are executed only once for each object in the dataset. Therefore, querying all the ranges takes $\mathcal{O}(n \log n)$ time. Finally, we have to traverse the range, check the distance, and update the answer. The time complexity of this stage depends on the objects in the range. The area of the range is $c \times 2\delta$, which is extremely small. It is reasonable to assume the objects are uniformly distributed in the small range. According to Eq. 1, the ratio of neighbors to non-neighbors in the range is $\pi/(4 - \pi)$. The average number of objects located in each range is $\frac{4}{\pi}\eta$. Hence, the time complexity of traversing ranges is $\mathcal{O}(n \times \frac{4}{\pi}\eta) = \mathcal{O}(n \times \eta)$. In conclusion, the time complexity of Algorithm 2 is $\mathcal{O}(n \log n + n \log n + n \times \eta) = \mathcal{O}(n \times \eta)$. The time complexity of Algorithm 3 is similar to Algorithm 2, which is the number of neighbors we have on the borders of partitions. Since the number of neighbors we have to explicitly output is $n \times \eta$, our algorithms are theoretically optimal.

4 GLADE

Given the aforementioned problems of the existing solution and the incapacity to accurately identify some highly-probable candidates, novel solutions have to be explored. The approach we take in this paper is a novel implementation of the real-or-bogus classifier in GLADE—a parallel data processing prototype we have developed from scratch over the past few years. Although we showed the considerable performance gains GLADE provides over PostgreSQL and Hadoop on a limited set of tasks focused mostly on SQL aggregate queries and analytics, including group-by aggregation, k-means clustering, and top-k ranking [Cheng et al. (2012)], we have not yet evaluated GLADE on a complex real-life application. Moreover, the characteristics of the PTF real-or-bogus classifier map perfectly on the GLADE architectural features. While these are compelling reasons to carry out such an investigation, the experimental results in Section 6 prove that GLADE is indeed a suitable solution that outperforms the existing PostgreSQL implementation.

GLADE. GLADE [Cheng et al. (2012); Cheng and Rusu (2013)] is a parallel data processing system specifically designed for the execution of analytical tasks specified in SQL enhanced with Generalized Linear Aggregates (GLA). This allows for the execution of a much larger class of analytical computations beyond the standard SQL aggregates. Essentially, GLADE provides an infrastructure abstraction for parallel processing that decouples the algorithm from the runtime execution. The algorithm has to be specified in terms of a clean interface – SQL + GLA – while the runtime takes care of all the execution details including data management, memory management, and scheduling. Contrary to existent parallel data processing systems designed for a target architecture, typically shared-nothing, GLADE is architecture-independent. It runs optimally both on shared-disk servers as well as on shared-nothing clusters. The reason for this is the exclusive use of thread-level parallelism inside a processing node while process-level parallelism is used only across nodes. There is no difference between these two in the GLADE infrastructure.

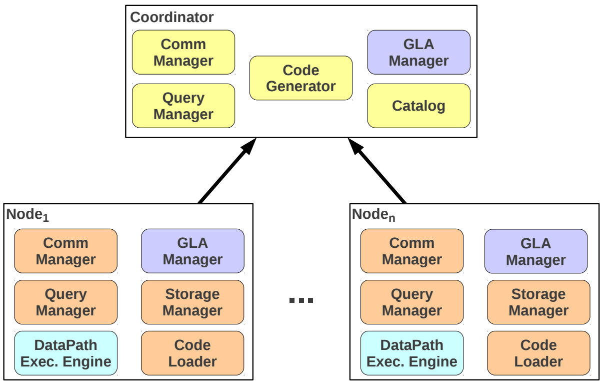

Figure 4: GLADE architecture.

Architecture. GLADE consists of two types of entities—a coordinator and one or more executor processes (Figure 4). The coordinator is the interface between the user and the system. Since it does not manage any data except the catalog metadata, the coordinator does not execute any data processing task. These are the responsibility of the executors, typically one for each physical processing node. It is important to notice that the executors act as completely independent entities, in charge of their data and of the physical resources. Each executor runs an instance of the DataPath [Arumugam et al. (2010)] relational execution engine enhanced with a GLA metaoperator for the execution of arbitrary user code specified using the GLA interface.

Communication Manager is in charge of transmitting data across process boundaries, between the coordinator and the executors, and between individual executors. Different inter-process communication strategies are used in a centralized environment with the coordinator and the executor residing on the same physical node and for a distributed shared-nothing system. The communication manager at the coordinator is also responsible for maintaining the list of active executors. This is realized through a heartbeat mechanism in which the executors send alive messages at fixed time intervals.

Query Manager is responsible for admission, setup, and query processing coordination across executors and queries. This is a particularly important task since processing is asynchronous both with respect to executors as well as to queries.

Code Generator fills pre-defined M4 templates with macros specific to the actual processing requested by the user generating highly-efficient C++ code similar to direct hard-coding of the processing for the current data. The resulting C++ code is subsequently compiled together with the system code into a dynamic library. This mechanism allows for the execution of arbitrary user code inside the execution engine through direct invocation of the GLA interface methods.

Code Loader links the dynamic library to the core of the system allowing the execution engine and the GLA manager to directly invoke user-defined methods. While having the code generator at the coordinator is suitable for homogeneous systems, in the case of heterogeneous systems both the code generator and the code loader can reside at the executors.

DataPath Execution Engine implements a series of relational operators – `SELECT`, `PROJECT`, `JOIN`, `AGGREGATE` – and a special `GLA` metaoperator for the execution of arbitrary user code specified using the GLA interface. They are all configured at runtime with the actual code to execute based on the requested processing. The execution engine has two main tasks—manage the thread pool of available processing resources and route data chunks generated by the storage manager to the operators in the query execution plan. Parallelism is obtained by processing multiple data partitions – chunks – simultaneously and by pipelining data from one operator to another.

GLA Manager executes `Merge` at executors and `Terminate` at coordinator, respectively. These functions from the GLA interface [Cheng et al. (2012)] are dynamically configured with the code to be executed at runtime based on the actual processing requested by the user. Notice that the GLA manager merges only GLAs from different executors, with the local GLAs being merged inside the execution engine.

Catalog maintains metadata on all the objects in the system such as table names and attribute names and their partitioning scheme. These data are used during code generation, query optimization, and execution scheduling. In addition to the global catalog, each executor has a local catalog with metadata on how its corresponding data partition is organized on disk.

Storage Manager is responsible for organizing data on disk, reading, and delivering the data to the execution engine for processing. The storage manager operates as an independent component that reads data asynchronously from disk and pushes it for processing. It is the storage manager rather than the execution engine in control of the processing through the speed at which data are read from disk. In order to support a highly-parallel execution engine consisting of multiple execution threads, the storage manager itself uses parallelism for simultaneously reading multiple data partitions.

Range-based data partitioning. Parallel execution is supported in GLADE through data partitioning, i.e., multiple partitions are processed simultaneously by different executors. The tuples of a relation are divided horizontally into chunks containing thousands to a few million tuples. Chunks are stored continuously on disk. The larger the size of the chunk, the longer the size of sequential scans, thus the smaller the number of disk seeks. While tuples can be assigned to chunks in arbitrary order, it is particularly useful for many workloads to have tuples with close values along some attributes grouped together in the same chunk. This corresponds to range-based data partitioning. Generating range-partitioned chunks is typically more complicated since data have to be partially ordered along the partitioning attributes. The benefit is faster execution for range queries since a reduced number of chunks have to be processed.

Column-oriented storage. Inside a chunk the columns of a relation are further partitioned vertically, with a disk page storing only values from the same column. Attribute values corresponding to the same tuple are stored at the same relative position inside each column. This allows for immediate tuple reconstruction in memory. The benefit of column-oriented storage is evident in the case of wide relations containing a large number of attributes with only a few of them accessed by every query. When range-based partitioning is combined with columnar storage – the case in GLADE – the amount of data read from disk is minimized since only the chunks and the columns required by the query are retrieved.

Multi-query processing. GLADE supports concurrent execution of multiple queries by sharing data access across the entire hierarchy—from disk to CPU registers through memory and cache. All the queries reading data from the same relation are connected to a single circular shared scan operator that reads a chunk only once and distributes it to all the queries that require it. While this is standard practice in any multi-query processing system, chunk sharing in GLADE is taken considerably further. Essentially, chunks are shared across all the common operators in the query execution trees corresponding to two queries. This requires merging separate operators with similar functionality into a single mega-operator that combines the operations corresponding to each query. For example, instead of having two selection operators with different predicates – one for each query – a single operator containing both predicates is created in GLADE. The new combined operator is responsible for identifying what queries a chunk is valid for and for setting the correct tuple validity based on the query predicates. The same logic can be applied to other relational operators, including `JOIN`, `GROUP BY`, `AGGREGATE`, and the `GLA` metaoperator. The code executed by each operator is dynamically generated at runtime based on the running queries.

Complex aggregates. The `GLA` metaoperator supports the execution of arbitrary user code specified using the abstract GLA interface [Rusu and Dobra (2012)]. This allows for the execution of complex computations far beyond standard SQL aggregates inside the execution engine without the need to extract data into an application with more powerful computational capabilities. This paradigm shift – bring the code near the data instead of moving data to the code – results in considerable gains especially in the cases where a large amount of data have to be moved.

5 Experimental Evaluation

In this section, we present the GLADE implementation for real-or-bogus classification. We show how data are mapped onto the GLADE storage model, how each task in the classification is expressed as GLADE computations, and how native GLADE features – range-based data partitioning, columnar storage, multi-query processing, and complex aggregate computation – are used in this workload. We provide measurement results that prove a significant improvement over the existing PostgreSQL solution and we analyze the reasons for this.

Data. The data we use in the experiments are a snapshot of the subtraction and candidate database. The 3 tables referenced in real-or-bogus classification and their characteristics are given in Table 1. The overall size of the 3 tables when loaded in GLADE is 161 GB. Notice that approximately 5,000 candidates are not classified by the random forest classifier—`rb_classifier` contains less tuples than `candidate`. There are 647 candidates per subtraction on average.
The maximum chunk size is fixed at $2^{20} \approx 1$ million tuples across all the tables. This generates a single `subtraction` chunk and 642 chunks for the other two tables. The size of a full chunk is different though across tables since they contain a different number of columns. Notice that only the columns required in query processing are read for a chunk – not the entire chunk – due to the columnar storage. Thus, even the `subtraction` table is never read in full unless all the columns are requested by the query. `candidate` and `rb_classifier` are range-based partitioned along the `subtraction_id` attribute. This guarantees that all the candidates extracted from the same subtraction end up in the same chunk. Moreover, candidates from subtractions close in time are also co-located in the same chunk with high probability. This partitioning has two benefits. It minimizes the number of chunks read from disk. And it isolates processing to the chunk level, thus increasing the amount of parallelism achieved by processing multiple chunks simultaneously.

Table name	Columns	Rows	Chunks
`subtraction`	51	1,039,758	1
`candidate`	46	672,912,156	642
`rb_classifier`	9	672,906,737	642

Table 1 Tables used in real-or-bogus classification.

5.1 Real-Time Pipeline

Setup. The machine used in the experiments is a low-end server with an Intel Core2 Quad CPU running at 2.66 GHz, 4 GB of memory, and a single 1 TB disk with sequential I/O throughput of 100 MB/s. Ubuntu SMP 10.10 64-bit is the operating system. There is a single GLADE executor in this configuration. It is co-located with the coordinator. Only thread-level parallelism is employed. The DataPath execution engine is configured to use 4 worker threads – one for each core – while the storage manager corresponding to every table assembles 4 chunks simultaneously. The reader might be surprised by our modest system choice given that the PTF pipeline is running in production on a powerful NERSC supercomputer. Nonetheless, our results confirm that even on such a low-end machine GLADE manages to load and classify the candidates in a set of subtractions in less than 20 minutes.

Data ingestion. The time it takes to ingest the entire dataset depicted in Table 1 in GLADE is 8,222 seconds ($\approx$ 2 hours 15 minutes) out of which 7,056 are spent loading the `candidate` table. This dataset corresponds though to many nights of observations. To determine how long it takes to ingest a set of subtractions generated at one instance in time, we chose a random night in the dataset, i.e., the night of October 10, 2011, compute its corresponding statistics, and then extrapolate the loading time based on these statistics. There are 2,997 subtractions taken during this night and 1,939,059 candidates at an average rate of 647 candidates per subtraction. The time taken to ingest these data into GLADE is only 24 seconds. Considering that the ingestion is distributed over 10 periods of 45 minutes each, the average ingestion time for a set of subtractions is less than 3 seconds.

Candidate identification. The first stage in real-or-bogus classification is to identify candidates with high realbogus score assigned by the random forest classifier. The corresponding query Q1 contains a 3-way join between `subtraction`, `candidate`, and `rb_classifier` and a series of selection predicates on each of the tables. The most important predicate is a range selection on `subtraction` that limits the search to the images acquired most recently. Due to range-based partitioning and columnar storage, in the GLADE implementation this query reads only the chunks and columns that generate results. In the optimal situation, a single chunk is processed from each of the 3 tables. Out of the almost 2 million candidates detected during the night of October 10, 2011, only 40,087 are classified as real by the random forest classifier. This is only 2%. The number can be easily increased by relaxing the conditions in the query. It takes GLADE only 9.2 seconds to find the real candidates, i.e., 0.92 seconds per subtraction.

```
Q1:
SELECT s.ujd, c.sub_id, c.id, c.ra, c.dec,
       c.xint_new, c.yint_new, c.pos_sub
FROM subtraction s JOIN candidate c
 ON (c.sub_id = s.id) JOIN rb_classifier
 rbc ON (rbc.sub_id = c.sub_id AND
         rbc.candidate_id = c.id)
WHERE (c.a_image < 3.0 OR c.mag < 15.0)
 AND s.ujd > 2455844 AND s.ujd < 2455845
 AND rbc.realbogus > 0.17
 AND rbc.bogus < 0.35
 AND c.b_image > 0.7
 AND c.pos_sub = 'True'
```

Candidate pruning. Each of the candidates identified by the random forest classifier is further checked before deemed real. The first condition a candidate has to pass is that it appears in more than one subtraction at a position close to the original position where it was first spotted. This is expressed as a complex nearest-neighbor query Q2 along the space and time attributes. In the current implementation of the PTF pipeline, one such query is executed sequentially for every candidate. This is 40,087 queries for our example night or approximately 4,000 queries for every subtraction set. Since all these queries have to be executed in less than 45 minutes, this step is by far the bottleneck of the entire process. GLADE multi-query processing kicks in perfectly in this situation thus allowing for multiple candidates to be checked at the same time. Most importantly though, the time to check many candidates – up to 64 candidates in the current GLADE implementation – is the same as checking a single candidate. The reason for this is that the queries have identical execution plans – more or less some constants – which allows for maximum data access sharing. When coupled with range-based partitioning and columnar storage, it takes only 18 minutes to check the 4,000 candidates identified in a subtraction set—18 seconds for a group of 64 candidates. 560 candidates survive pruning on average for a subtraction set.

```
Q2:
SELECT COUNT(*)
FROM subtraction s JOIN candidate c
 ON (c.sub_id = s.id) JOIN rb_classifier
 rbc ON (rbc.candidate_id = c.id)
WHERE c.ra BETWEEN (%1f,%2f)
 AND c.dec BETWEEN (%3f,%4f)
 AND (s.ujd BETWEEN (%5f,%6f) OR
      s.ujd BETWEEN (%7f,%8f))
 AND (rbc.realbogus > 0.07 OR
      c.pos_sub <> 'True')
 AND c.b_image > 0.7
 AND (c.a_image < 3.0 OR c.mag < 15.0)
```

Contextual realbogus computation. For the surviving candidates, the contextual realbogus score is computed based on the probability of being real of their nearest-neighbor candidates in the subtraction. This requires another iterative process in which each surviving candidate is examined independently. The difference from pruning is that the contextual realbogus score cannot be computed inside the database. Instead it is computed in a Python script that extracts the necessary data from the database using query Q3. This is not required in GLADE though since complex aggregates can be expressed as GLAs and executed inside the system without moving data between processes. In addition to the savings in execution time, the GLA mechanism allows for all the computation to be confined to the database engine—a cleaner and easier to understand solution. In GLADE, the contextual realbogus score for the 560 candidates surviving pruning in a subtraction set is computed in 88 seconds—it takes 10 seconds on average to compute the score for a group of 64 candidates.

Table 2 summarizes the results we obtained for processing the October 10, 2011 data in GLADE. These are average results for processing a subtraction set. The overall time to classify the candidates is less than 20 minutes. This is less than half the length of the interval between two sets of images are ingested, i.e., 45 minutes. The remaining time can be used either to increase the rate at which images are ingested or to analyze more candidates—some of the conditions based on which candidates are pruned are arbitrary and they are targeted at reducing the overall classification time. This is not a problem in the GLADE implementation though.

```
Q3:
SELECT c.id,
  $\sqrt{(c.xint\_new - \%1f)^2 + (c.yint\_new - \%2f)^2}$ AS dist
FROM subtraction s JOIN candidate c
 ON (c.sub_id = s.id) JOIN rb_classifier
 rbc ON (rbc.sub_id = c.sub_id AND
        rbc.candidate_id = c.id)
WHERE s.id = %3i AND c.pos_sub = '%4s'
ORDER BY dist
```

PostgreSQL solutions. In order to compare the proposed GLADE approach with the existent solution, we devise two alternative PostgreSQL databases. The first database does not contain any optimizations. There are no indexes or any other structures for enhancing query performance. The second database defines indexes for all the attributes used in selection predicates or join conditions across the three workload queries. This is the solution implemented in the PTF production pipeline. We deploy these two databases on a PostgreSQL 8.4 server running on the same test machine. We modify the server configuration in order to maximize usage of the available memory resources in the system, e.g., we set `shared_buffers` to 3 GB.

Table 2 contains the results for the two PostgreSQL database implementations. The indexed implementation outperforms the non-indexed version considerably at query processing. The gap is as much as 6 orders of magnitude for the contextual realbogus computation. The reason for this is that the non-indexed database has to read all data from all the tables in order to perform any query. Since no indexes are available, sequential scan is the only feasible path access strategy. Indexes reduce dramatically query execution time since the tuples satisfying the highly-selective predicates can be identified with as little as a single disk access. Due to the large buffer pool, disk access is not even required at all in many situations. Indexes also play an important role in the selection of the join algorithms used in query execution plans. The situation is radically different though for data ingestion. While it takes less than a minute to load a new set of candidates in the non-indexed database, it takes 68 minutes to do so in the indexed version. This 61 factor difference is entirely due to index maintenance. Adding tuples to the `candidate` table requires insertions in each of the 9 indexes defined over it. Although this might not seem such a difficult problem at first, in the case of a batch of 200,000 insertions the probability to encounter some time-consuming index reorganizations is quite high.

Overall, none of the PostgreSQL solutions meets the requirement to ingest and identify a new set of candidates in less than 45 minutes. The index-based solution deployed in production takes 75 minutes on our test machine—out of which 60 minutes are spent for data ingestion. The non-indexed version is far from this requirement. A possible solution to decrease the loading time for the indexed database is to reduce the number of indexes. The expectation is that the decrease in loading time offsets the increase in query execution time and for some combination of indexes the overall time drops below 45 minutes. Finding the optimal index combination is a hard problem that requires the investigation of an exponential number of alternatives.

Observations. When we compare the proposed GLADE solution to the PostgreSQL indexed database, we remark some interesting aspects. Overall, GLADE outperforms PostgreSQL by a factor of 3.88. This is entirely due to the efficient GLADE data loading mechanism which is faster by 3 orders of magnitude. Since GLADE does not employ any secondary data structures to enhance query performance, it is not as efficient as indexed PostgreSQL in answering queries. The difference between the two systems – only 24% – is considerably smaller when compared to the basic PostgreSQL implementation. The GLADE architecture specifically targeted at analytical processing and optimized for read-mostly workloads is responsible for providing similar query performance to indexed PostgreSQL but without the associated increase in database size – the indexed PostgreSQL database is twice as large as GLADE – and ingestion time—GLADE loads new candidates a factor of 60 faster. Range-based partitioning and columnar storage combine together in order to minimize the amount of data read from disk across all types of range queries. Dedicated support for the execution of any user code inside the system eliminates data movement almost entirely and allows for complex computations to be executed right near the data. For the PTF workload though, the most significant gains are due to multi-query processing. Instead of verifying each candidate one at a time, GLADE allows for up to 64 candidates to be evaluated simultaneously in the same amount of time. This is because all the queries have identical execution plans and GLADE is capable to combine them into a single plan in which the operators share access along the entire data path—from disk to CPU registers through main memory and cache.

5.2 *Secondary Analysis*

Setup. The secondary analysis experiments are executed on Edison – a Cray XC30 supercomputer at NERSC [Edison (2016)] – with peak performance 2.57 PFlops/second on 5,576 compute nodes, 133,824 cores in total. Nodes are linked by a high-speed interconnect with Dragonfly topology having ≈8 GB/sec MPI bandwidth. The aggregate memory of Edison is 357 TB and the scratch storage capacity is 7.56 PB. In our experiments, we use 64 nodes which provide 3,072 hyper-threads. The distance parameter c is set to 0.001 degrees.

Implementation. An efficient file partitioner is implemented in `C++`. It takes the desired number of partitions as input and computes a coarse partition step size. Each thread is assigned the corresponding partition and is in charge of accessing it concurrently with the other threads. Algorithms 2 and 3 are implemented as a GLADE GLA. The state of the GLA consists of the balanced tree T—one for each partition. Partition processing is implemented in the `Accumulate` method, while merging in `Merge`. At the end of processing a partition, only the objects at the boundaries are saved in the GLA state for merging with the neighboring partitions. We use the `C++ STL set` as the balanced tree T. *range* is implemented as function calls to *set::lower_bound* and *set::upper_bound* which run in $\mathcal{O}(\log n)$ time, where n is the size of T. The running time of *range* depends on the number of objects located between the lower and upper bound.

PostgreSQL solution. *q3c* [Koposov and Bartunov (2006)] is a package for PostgreSQL that implements a quad-

Phase	GLADE	PostgreSQL	PostgreSQL + indexes
Data ingestion	3 sec	59 sec	1 hour 8 sec
Identification	0.92 sec	45 sec	4.67 sec
Pruning	18 min	607 hours	15 min 39 sec
Contextual realbogus	1 min 30 sec	68 hours	0.79 sec

Table 2 Average results for processing a subtraction set on October 10, 2011.

tree index. After building the index with *q3c*, we can find the neighbors of an object by executing its corresponding self-join SQL query in Section 3. Our goal is to execute the query for each object in the partition. *q3c* does not provide support for concurrent access beyond the standard PostgreSQL mechanism and is, thus, not scalable. Therefore, we can only compare the average query execution time of the proposed solution with that of PostgreSQL for a single query.

Phase	GLADE	PostgreSQL + q3c
Computation	290 s	N/A
Merge	10 s	N/A
Total	300 s	N/A
Average per object	0.004 ms	20 ms

Table 3 Execution time (in seconds) for secondary analysis.

Results. Table 3 depicts the results for GLADE and PostgreSQL. The GLADE solution completes the computation for all the partitions in 290 seconds. Merging the borders takes 10 seconds, for a total of 300 seconds. The average execution time per object in GLADE is, thus, 0.004 ms. The average execution time in PostgreSQL on 1 million random objects is 20 ms—a factor of 5000 larger. We can observe that GLADE achieves more than 3 orders of magnitude improvement over the optimized PostgreSQL solution with *q3c*. The batch processing and lightweight balanced tree significantly reduce the computation time. The batch processing mechanism takes advantage of the intermediate results for neighboring objects. In addition, the balanced tree contains only objects whose *dec* is no larger than the current *dec* − *c*, unlike *q3c* which builds the index over all the objects. Therefore, our algorithm runs much faster when answering lower bound and upper bound queries, which is the bottleneck of the *q3c* solution.

6 Conclusions

In this paper, we present a novel implementation for the real-or-bogus classification in GLADE—a parallel multi-query processing system targeted specifically at analytical workloads. We show how each stage in the classifier – candidate identification, pruning, and contextual realbogus – maps optimally into GLADE tasks by taking advantage of the unique features of the system—range-based data partitioning, columnar storage, multi-query execution, and in-database support for complex aggregate computation. The result is an efficient classifier implementation capable to process a new set of acquired images in a matter of minutes even on a low-end server. For comparison, the existing optimized PostgreSQL implementation of the classifier is a factor of 3.88 slower. Due to this reduction in the time to investigate a set of new candidates, considerably more candidates can be thoroughly evaluated, thus increasing the likelihood to find many transients that are otherwise missed by the current solution.

In addition, we introduce a novel parallel similarity join algorithm for advanced transient classification. This algorithm operates offline and considers the entire candidate dataset consisting of all the objects extracted over the lifetime of the Palomar Transient Factory survey. We implement the similarity join algorithm in GLADE and execute it on a massive supercomputer with more than 3000 threads. We achieve more than 3 orders of magnitude improvement over the optimized PostgreSQL solution.

Our evaluation over two very different computational architectures – a multi-core server and a distributed cluster – confirm that GLADE is capable of optimally exploiting the available resources and maximizing performance for the PTF workload. In the future, we plan to apply GLADE to other scientific applications that require diverse computational resources, such as financial services [Duran et al. (2014)], weather modeling [Almeida et al. (2016)], and earth modeling [Wang et al. (2016)].

Acknowledgments. This work is supported in part by the Director, Office of Laboratory Policy and Infrastructure Management of the U.S. Department of Energy under contract No. DE-AC02-05CH11231 and by a U.S. Department of Energy Early Career Award (DOE Career). We thank the anonymous reviewers for their insightful comments that improved the quality of the paper significantly.

References

Almeida, E., Bauer, M. A. and Fazenda, A. L. (2016) "Numerical Weather Model BRAMS Evaluation on Many-Core Architectures: A Micro and Macro Vision", *International Journal of Computational Science and Engineering*, Vol. 12, No. 4, pp. 330–340.

Arumugam, S., Dobra, A., Jermaine, C., Pansare, N. and Perez L. (2010) "The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses", *International Conference on Management of Data ACM SIGMOD 2010*, pp. 519–530.

Bloom, J. S. et al. (2011) "Automating Discovery and Classification of Transients and Variable Stars in the Synoptic Survey Era", *CoRR*, abs/1106.5491.

Cheng, Y., Qin, C. and Rusu, F. (2012) "GLADE: Big Data Analytics Made Easy", *International Conference on Management of Data ACM SIGMOD 2012*, pp. 697–700.

Cheng, Y. and Rusu, F. (2013) "Astronomical Data Processing in EXTASCID", *International Conference on Scientific and Statistical Database Management SSDBM 2013*, pp. 387–390.

Das Sarma, A., He, Y. and Chaudhuri, S. (2014) "ClusterJoin: A Similarity Joins Framework using MapReduce", *PVLDB*, Vol. 7, No. 12, pp. 1059–1070.

Duran, R., Chen, D., Saraswat, R. and Hallmark, A. (2014) "A Framework for Comparing High Performance Computing Technologies", *International Journal of Computational Science and Engineering*, Vol. 9, No. 1/2, pp. 119–129.

Edison@NERSC `http://www.nersc.gov/users/computational-systems/edison/` [Last accessed: June 2016]

Grillmair, C. J. et al. (2010) "An Overview of the Palomar Transient Factory Pipeline and Archive at the Infrared Processing and Analysis Center", *ASP Conf. Ser. 434, Astronomical Data Analysis Software and Systems XIX*, pp. 28–36.

Koposov, S. and Bartunov, O. (2006) "q3c", *Astronomical Society of the Pacific Conference Series*, Vol. 351, pp. 735–750.

Law, N. M. et al. (2009) "The Palomar Transient Factory: System Overview, Performance and First Results", *CoRR*, abs/0906.5350.

Palomar Transient Factory, `www.ptf.caltech.edu/iptf/` [Last accessed: June 2016]

PostgreSQL, `http://www.postgresql.org/` [Last accessed: June 2016]

Python Programming Language, `http://www.python.org/` [Last accessed: June 2016]

Rusu, F. and Dobra, A. (2012) "GLADE: A Scalable Framework for Efficient Analytics", *Operating Systems Review*, Vol. 46, No. 1, pp. 12–18.

Rusu, F., Nugent, P. and Wu, K. J. (2014) "Implementing the Palomar Transient Factory Real-Time Detection Pipeline in GLADE: Results and Observations", *International Workshop on Databases in Networked Information Systems DNIS 2014*, pp. 53–66.

Wang, D., Domke, J., Mao, J., Shi, X. and Ricciuto, D. M. (2016) "A Scalable Framework for the Global Offline Community Land Model Ensemble Simulation", *International Journal of Computational Science and Engineering*, Vol. 12, No. 1, pp. 73–85.