Sharing Work in Keyword Search over Databases

Marie Jacob
University of Pennsylvania
Philadelphia, PA, USA
majacob@cis.upenn.edu

Zachary Ives
University of Pennsylvania
Philadelphia, PA, USA
zives@cis.upenn.edu

ABSTRACT

An important means of allowing non-expert end-users to pose ad hoc queries — whether over single databases or data integration systems — is through keyword search. Given a set of keywords, the query processor finds matches across different tuples and tables. It computes and executes a set of relational sub-queries whose results are combined to produce the k highest ranking answers.

Work on keyword search primarily focuses on single-database, single-query settings: each query is answered in isolation, despite possible overlap between queries posed by different users or at different times; and the number of relevant tables is assumed to be small, meaning that sub-queries can be processed without using cost-based methods to combine work. As we apply keyword search to support ad hoc data integration queries over scientific or other databases on the Web, we must *reuse and combine* computation. In this paper, we propose an architecture that continuously receives sets of ranked keyword queries, and seeks to reuse work across these queries. We extend multiple query optimization and continuous query techniques, and develop a new query plan scheduling module we call the ATC (based on its analogy to an air traffic controller). The ATC manages the flow of tuples among a multitude of pipelined operators, minimizing the work needed to return the top-k answers for all queries. We also develop techniques to manage the sharing and reuse of state as queries complete and input data streams are exhausted. We show the effectiveness of our techniques in handling queries over real and synthetic data sets.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Performance, Management, Algorithms

Keywords

Keyword search, top-k queries, multiple query optimization

1. INTRODUCTION

In recent years, there has been increasing acknowledgment that database technologies must be made more accessible to end-users. One promising approach is that of providing keyword search interfaces over tables or XML trees [1, 2, 11, 12, 13, 18, 27, 33]: here, keyword searches get reformulated into sets of database-style join queries, whose answers are merged and ranked according to some distance, cost, or relevance metric. Keyword search interfaces are especially appealing for settings where users must pose novel (ad hoc) queries over large numbers of tables that they do not fully understand: this occurs, for instance, in scientific communities where many interrelated public databases, rather than a single global data warehouse, exist. A specific example is the life sciences, where individual databases may have more than 350 tables [21], and multiple such databases may have relevant data that can be pieced together to answer a scientist's exploratory query.

Anecdotally, scientists often pose *sequences* of related queries, iteratively refining their search for a particular relationship; and they may refer back to previous queries[1]. Moreover, certain core concepts (e.g., proteins) will appear in many queries.

Unfortunately, while a substantial body of research has been conducted on keyword search over databases, such work has been directed at settings in which a *single database* is answering a *single user's query* in isolation. Techniques have been developed to use precomputed indices to quickly winnow down the number of database subqueries that must be posed to return the top query answers [9, 13], to find promising join "paths" linking the matching tables [1, 2, 11, 12, 13, 18, 27, 33], and to efficiently compute and merge top-k results [7, 28]. These techniques assume that limited numbers of join subqueries are necessary to obtain top-k answers, and that few relations appear in multiple join subqueries. In the data integration setting, it is much more common that *multiple* source relations have highly relevant answers for each keyword; we must consider ways of computing all combinations of these relations, thus computing a set of queries with high mutual overlap (see, e.g., [33]). Moreover, as mentioned previously, we may wish to exploit the fact that queries posed across time may overlap.

In this paper we focus on scaling keyword search interfaces to data integration scenarios with commonly-used sources, by exploiting the overlap among the join subqueries that partially answer a *single* keyword search, and also the overlap among *different* keyword searches performed over time and even across users. We show that through cost-based computation of common subexpressions, clever caching, and pruning of the query optimization search space, we can handle large numbers of queries and relevant sources — while still delivering top-k results quickly.

We assume a very general model of ranked query processing, in which the score of a query result may include a *static* compo-

SIGMOD'11, June 12–16, 2011, Athens, Greece.

[1]See, e.g., Protein Data Bank, `pdbbeta.rcsb.org/robohelp/site_navigation/reengeneered_site_features.htm`, for a system that supports query reuse.

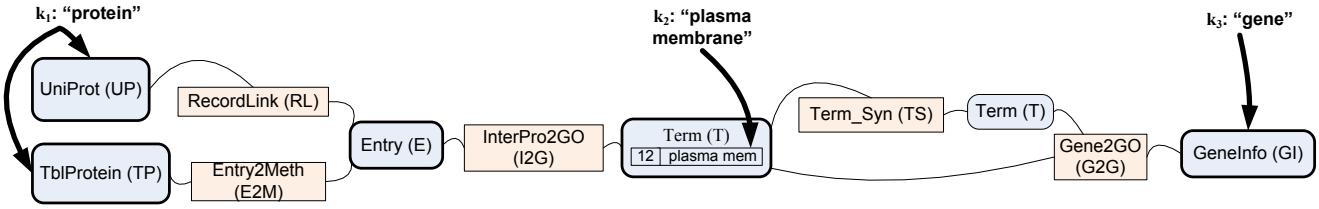

Figure 1: Example of a set of relations from multiple bioinformatics databases, plus a keyword query, for which each keyword ($k_1 \ldots k_3$) may match a table either based on its name, or based on an inverted index of its content.

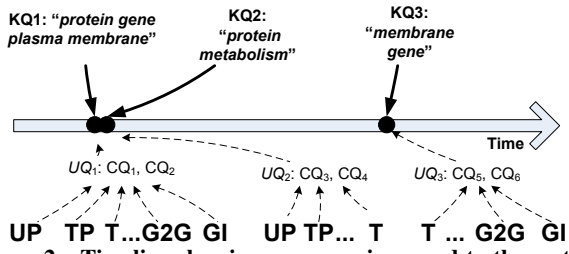

Figure 2: Timeline showing user queries posed to the system. Note the overlap between tables that they access.

nent (based on the formulation of the query and/or properties of the source relations) as well as a *dynamic* one (based on specific attribute values from the source tuples). We even allow for the fact that different users may have different scoring functions, possibly taking into account context or user preference [3, 20, 31, 33]. We assume that keyword searches are posed over time — sometimes concurrently, sometimes in sequence. When queries are posed concurrently, we perform cost-based multiple query optimization over the query batch. When they are posed over time, we reuse results computed by the earlier queries (resembling a continuous query model [4]). See Figures 1 and 2 for an example database schema and a visualization of queries posed over time (we describe details in Section 2). Our approach takes advantage of several properties: (1) top-k queries generally create a limited amount of intermediate state [16], as they only run until the desired number of results are processed; (2) it is often possible to compute bounds on the range of scores for tuples from any query, by considering the query's static score component [33] and information about score attribute maxima; (3) top-k queries should support fully pipelined execution so each subquery only reads the amount of data necessary to produce top-scoring answers; (4) even subqueries that use different scoring functions will read from the source relations in the same order — they will just read at different relative rates from the sources.

Our work is implemented as part of the Q System [32, 33], which enables customization of ranking functions for each user. It could alternatively be used with the Kite data integration system [27] or with other keyword search in database schemes [2, 11, 13]. We make the following contributions:

- A new top-k query processing architecture that keeps in-memory state and existing query plan structures from one query execution to the next, enabling effective result reuse and recomputation over time.
- An optimizer that combines cost-based and heuristic search, in order to find common subexpressions across and within top-k queries, and which supports the reuse of existing subexpressions from past query executions.
- A fully pipelined, *adaptive* top-k query execution scheme for answering multiple queries, consisting of a query plan graph of m-joins (STeM eddies) [24, 34] and rank-merge [7] operators, supervised by a novel ATC controller.
- A query state manager that can *graft* and *prune* elements in an executing query plan, and which manages the reuse and eviction of state.
- A comprehensive set of experiments over synthetic and real datasets, demonstrating the performance gains of our approach.

We provide a problem definition in Section 2, then outline our approach in Section 3. Section 4 explains how multiple queries can be executed simultaneously. We describe how we optimize batches of queries for simultaneous execution in Section 5, then how we handle further queries over time in Section 6. We experimentally validate the performance of our system in Section 7, we describe related work in Section 8, and we conclude and describe future work in Section 9.

2. PROBLEM DEFINITION

Our goal in the Q System is to provide high performance to a user base that continuously poses keyword queries over remote (and possibly local) database instances — where the resulting top-k SQL queries overlap in the relations they access, both with other concurrent queries and with other queries to be posed in the future. Intuitively, the goal is to create and maintain a small set of pipelined *query plan graphs* consisting of query processing operators — where each operator may produce sub-results that get fed to one *or more* downstream operators, ultimately depositing results into *ranking queues* that return the top-k answers. In general, we expect the plans' in-memory state to be small, if the queries have a high overlap with one another and can share work. Moreover, top-k queries tend to only fetch a small number of the highest-scoring tuples from their relations. Once execution of a batch of queries completes, we do not discard the query plan graph and its state — rather, we take subsequent queries and attempt to *graft* them onto the existing graph, reusing existing plan state and operators as appropriate. Only if we run out of resources (in particular, memory) do we *prune* plan state and operators.

2.1 Keyword Search in Data Integration

We introduce some notation and review the basics of keyword search over databases, with an emphasis on databases that may be remotely stored and in need of integration, as in [27, 33].

EXAMPLE 1. *Suppose we are running a portal for biologists to pose ad hoc queries. Perhaps we have a known schema graph, such as the one shown in Figure 1. The sources UniProt, ProSite and InterPro correspond to protein databases; GeneOntology, NCBI Entrez correspond to gene information, and OMIM stores information about genetic diseases. These tables are bridged by record linking tables (shown as orange squared rectangles). Edges in the graph represent foreign keys, hyperlinks, and potential join relationships.*

A biologist may pose the keyword query KQ_1*: "protein 'plasma membrane' gene." A keyword search system would find matches between each keyword and data/metadata of each relation (as shown*

Table 1: Conjunctive queries CQ_1, CQ_2 from user query UQ_1, answering keyword query KQ_1.

CQ_1	*q*(*prot*, *gene*, typ, *dis*):-*TP*(*id*, *prot*, ...), *E2M*(*ent*, *id*, ...), *I2G*(*ent*, *gid1*), **T**(*gid1*, 'plasma membrane', *score*), **TS**(*gid1*, *gid2*, *score*), *G2G*(*gid2*, *giId*), *GI*(*giId*, *gene*, ...)
CQ_2	*q*(*nam*, *gene*, *typ*, *dis*):-*UP*(*ac*, *nam*, ...), **RL**(*ac*, *nam*, *ent*, *prot*, *score*), *I2G*(*ent*, *gid1*), **T**(*gid1*, 'plasma membrane', *score*), *G2G*(*gid1*, *giId*), *GI*(*giId*, *gene*, ...)

Table 2: Conjunctive queries CQ_3, CQ_4 from user query UQ_2, answering keyword query KQ_2, posed at the same time as KQ_1.

CQ_3	*q*(*prot*, *typ*):-*TP*(*id*, *prot*, ...), *E2M*(*ent*, *id*, ...), *I2G*(*ent*, *gid1*), *T*(*gid1*, 'metabolism', *score*)
CQ_4	*q*(*nam*, *typ*):-*UP*(*ac*, *nam*, ...), **RL**(*ac*, *nam*, *ent*, *prot*, *score*), *I2G*(*ent*, *gid1*), **T**(*gid1*, 'metabolism', *score*)

in Figure 1). In some cases, the match might be exact while in other cases, it might be approximate. The goal of the keyword search system is find a set of trees that contain relations matched to all keywords. Each subgraph is then mapped to a conjunctive query which is then evaluated. Table 1 shows conjunctive queries CQ_1 and CQ_2 corresponding to our example. The answers to KQ_1 may include results from these and other conjunctive queries; together these form the user query UQ_1. □

Keyword searches are inherently uncertain during the matching process, and systems will typically score the answers according to some score function C that maps a tuple answer to a real score value. Many scoring models have been proposed, with most usually taking into account a static component (such as query size), as well as a dynamic component (such as keyword-to-tuple score matches). We describe three different models/systems that we believe are representative of the state of the art:

DISCOVER: The systems in [12, 13] enumerate ordered conjunctive queries (called *candidate networks*) for a particular keyword search, and rank tuples based on the size of the query that produced the tuple, as well as standard IR scores that are derived from a particular DBMS. Under their ranking model, a sample score function C for CQ_1 would be $C(t) = \frac{1}{size(CQ_1)}$ or $C(t) = \frac{\sum_{t_i} score(t_i)}{size(CQ_1)}$, where t is a tuple of CQ_1 and t_i belongs to the derivation of t.

Q System: The model in the Q System [32, 33] is similar to that of DISCOVER, in that conjunctive queries are found over a schema graph. Each edge in the graph is annotated with a cost that represents how useful the edge is. Additionally, each relation may be annotated with a cost that denotes how authoritative it is. These edge and node costs are learned and may be different across user queries. For a result tuple t of CQ_1, the scoring function C would be $C(t) = \frac{1}{2^c}$, where $c = (\sum_e c_e) + (\sum_i cost(t_i))$, c_e is a cost for each edge in the subgraph and $cost$ is a function that maps a tuple to a cost value.

BANKS, BLINKS: Unlike the previous systems, query execution in BANKS [2] and BLINKS [11] involves traversing a *data* graph (where nodes are tuples and edges are key/foreign key relationships between tuples), in order to enumerate the top-k list. Nodes and edges in the graph are annotated with scores and weights, with the ranking function forming a monotonic combination of both.

The challenge for keyword search systems lies in efficiently returning the top-k answers, *without* exhaustively computing every answer satisfying the query conditions. If we know the potential range bounds of scores for the conjunctive queries, we may be able to eliminate some queries from consideration, because they will not return top-scoring answers. If we fetch tuples from the various relations in decreasing order of their score, we may likewise be able to stop once we have returned enough answers. These concepts have formed the cornerstones of work on top-k query processing, ranging from keyword search systems [2, 11, 13, 18, 19, 33] to generic rank-merging algorithms [7] to custom rank-join algorithms [5, 9, 12, 15, 16, 23, 28] and ranked query optimization strategies [22]. Our goal is to build on the foundations of all these works, to address (for any of the three scoring models) two tasks that can arise from the keyword search scenario of Figure 2.

2.2 Task 1: Handling overlapping queries

EXAMPLE 2. *Concurrently with the first biologist, a second biologist may simultaneously pose a second keyword query KQ_2 to the system: "protein metabolism." Two example conjunctive queries are shown in Table 2. Observe that these queries not only overlap with each other — but they are also subexpressions of CQ_1 and CQ_2, respectively. Note, however, that KQ_2 was posed by a different user, and so a* different scoring function *might be associated with this query (our Q System supports custom ranking functions for each user [33]).*

This overlap in concurrently posed queries provides an opportunity for doing *shared computation*, which has only been explored to a limited extent in the keyword search context. The works of [2, 11] investigate this to some degree, where sharing computation essentially amounts to discovering join paths from a single (keyword-matched) tuple to other tuples in a data graph. They thus employ graph searching algorithms, assuming that the relationships between tuples are already known, and that the cost of joining two tuples (i.e., traversing a link) is negligible. We address a more general case, where tuples are stored in remote databases on the Internet, which can be accessed in streaming fashion or *probed* (remotely queried for matches) using some join key.

Our **first set of contributions** in this paper is a new scheme for simultaneously producing results for multiple pipelined, ranked queries within a middleware layer for data integration. We emphasize pushing down subqueries, whose results get streamed to the middleware, where they are joined with one another, and with sources that must be remotely probed. It is important to *simultaneously* (in interleaved fashion) support answering of *multiple* conjunctive queries: Such queries might be ones whose results are being merged in rank order to form top-k answers, as with queries CQ_1 and CQ_2 in our previous example; or they might be returning results to *different* users whose queries are posed simultaneously. In our context, this requires fully *pipelined* execution of joins. To share work across query plans, we support *graph-structured* rather than tree-structured query plans, where a given query subexpression may produce answers whose results must be fed into multiple "downstream" operators belonging to different queries.

2.3 Task 2: Handling dynamic changes

Our interest is not only in efficiently executing sets of queries posed concurrently, but also in handling the *dynamic* operation of the system: over time new users pose new queries, and existing users *refine* their keyword queries by posing related queries with further or fewer keywords. Recall the scenario of Figure 2.

Table 3: Two conjunctive queries CQ_5, CQ_6 from user query UQ_3, corresponding to the first user's second keyword query, KQ_3.

CQ_5	*q*(*gene*, *typ*, *dis*):-**T**(*gid*, 'plasma membrane', *score*), *G2G*(*gid*, *giId*), *GI*(*giId*, *gene*, ...)
CQ_6	*q*(*gene*, *typ*, *dis*):-**T**(*gid1*, 'plasma membrane'), **TS**(*gid1*, *gid2*, *conf*), *G2G*(*gid2*, *giId*), *GI*(*giId*, *gene*, ...)

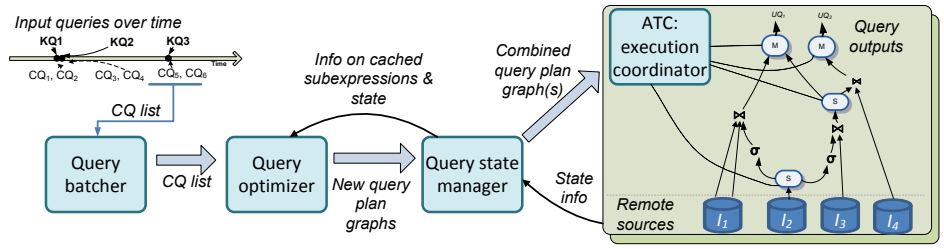

Figure 3: Query processing components of the Q System. Sets of CQs from the query generator are batched, optimized to exploit state from prior executions, and *grafted* into running query plan graphs for execution by the ATC.

EXAMPLE 3. *Suppose the first user, upon getting the results for KQ_1, decides to* revise *or refine his or her query, in this case to see what other genes are related to the plasma membrane. This results in a new keyword query KQ_3, "membrane gene". The resulting user query UQ_3 includes conjunctive queries CQ_5 and CQ_6 in Table 3. Observe that both of these queries are subexpressions of the original CQ_1, omitting the protein-related relations.* □

Perhaps our system has completed execution of the query UQ_1 from the first example; but if we retain its state (and portions of its query plan) in memory, we might be able to *reuse* portions of the state and plan to answer UQ_3. Our **second set of contributions** focus on retaining a query plan graph and reusing it from one query execution step to the next, with appropriate "grafting" of new operators into the existing plan, and "pruning" of operators and state as necessary to fit within a memory budget.

3. APPROACH AND ARCHITECTURE

Our presentation focuses on the modules *after* keyword terms have been converted to ranked lists of candidate conjunctive queries (candidate networks): we assume a set of conjunctive queries for each search, generated using any of the methods cited in Section 2.1[2].

Formally, each *keyword query* KQ_j is converted to the union of a set of conjunctive queries: we refer to this set as UQ_j. Each conjunctive query CQ_i within UQ_j is paired with a monotonic *score function* C_i, which maps result tuples to real-valued scores, following any of the cost models of Section 2.1. We assume the existence of a function $\mathbf{U}(C_i)$ that allows us to compute the upper bound on the score of any tuple returned by CQ_i. Finally, we assume the conjunctive queries return results in nonincreasing order relative to **U**. We assume the source relations referenced in the queries are typically SQL DBMSs, able to return results in nonincreasing score order; we refer to such sources as *streaming sources*. In addition to streaming sources, we expect other Web-based sources that can be *probed* with specific tuple values to return join results [10, 25]. We refer to these as *random access sources*. We now describe how our implementation, within the Q System, processes these queries.

Refer to Figure 3 for a diagram of the query processing architecture of the Q System, which functions as a middleware layer over remote data sources. Users pose keyword queries KQ_j that get converted into conjunctive queries and associated scoring functions: these are passed along to the **query batcher** as lists of triples $[(UQ_j, CQ_i, C_i)|CQ_i \in UQ_j]$, in nonincreasing order of the maximum score they return, i.e., their bound $\mathbf{U}(CQ_i)$. The batcher typically waits for these conjunctive queries to collect over a small time interval before it passes them along to the next stage. Following our example, the list of triples after keyword queries KQ_1 and KQ_2 would be $[(UQ_1, CQ_1, C_1), (UQ_1, CQ_2, C_2), (UQ_2, CQ_3, C_3), (UQ_2, CQ_4, C_4)]$.

The **query optimizer** takes a set of conjunctive queries from the batcher, and performs several forms of multiple query optimization over them. It first checks with the **query state manager** to determine if previous queries have been executed and their runtime state has been preserved: if this is the case, it determines what query expressions can be reused from in-memory buffers. Then it finds an efficient plans for computing the complete set of batched queries (which may involve reusing *or* recomputing existing expressions, sharing computation across multiple conjunctive queries in the batch, or pushing down portions of the query to the remote data sources). For example, the optimizer may note the high similarity in the new conjunctive queries of UQ_3 to those of UQ_1 and UQ_2, and it may reuse tuples that have already been read in for some of the common relations. The result of this optimization is one or more *query plan graphs*, which form the core of our query processor. We discuss query plan graphs and operators in the next section. Briefly, each plan graph may share subexpressions across queries by pipelining results through a *split* operator; each conjunctive query joins multiple relations using an *m-join* operator; conjunctive queries' results are merged into the top-k answers to a user query via a *rank-merge* operator. Execution of these operators is coordinated by a module called the **ATC**, named after an air traffic controller, as it routes tuples to different destination operators.

The **query state (QS) manager** is responsible for managing the set of query plan graphs that occupy the CPU and memory. Given a new set of query plan graphs, its first task is to *merge* them with any plan graphs that currently exist in memory. For each user query, it *incrementally* takes the highest-scoring conjunctive queries (which come in nonincreasing order of maximum-scoring contribution to the results) and identifies which subexpressions overlap with existing plans; it then *grafts* the new subgraphs into place (using *split* operators) and notifies the execution system. As execution progresses and the maximum score of the next result drops, further conjunctive queries can be activated. Additionally, after producing some answers, certain queries may no longer be able to contribute to top-k results: the QS manager deactivates (*prunes*) these for efficiency. Even if a query subexpression gets pruned from execution, its state is retained for possible reuse, until the system runs out of cache space and needs to evict it. Finally, the QS manager maintains cardinality information about intermediate results in the query plan graph, such that the query optimizer can determine what can be reused in subsequent executions.

[2]BANKS and BLINKS enumerate tuples instead of queries, but one could abstract their data graph into a schema graph.

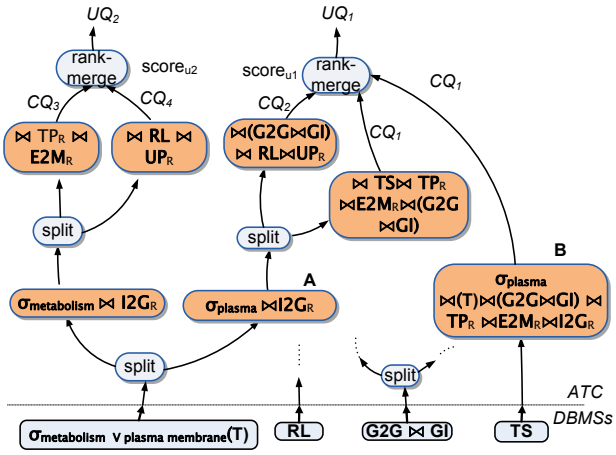

Figure 4: Partial query plan graph demonstrating *split*, *rank-merge* and *m-join* operators.

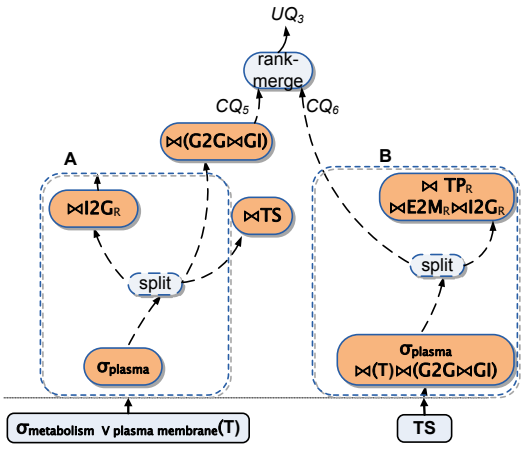

Figure 5: Grafting new segments within the marked modules from Figure 4 by using new split operators.

We now describe our simultaneous query execution process in more detail, deferring a discussion of optimizing query batches to Section 5 and of reusing results to Section 6.

4. QUERY PLAN GRAPH

Unlike in a traditional iterator-based query processor, our goal here is to fully pipeline and multiplex the execution of *multiple queries simultaneously*. In addition, we seek to preserve an important aspect of top-k query processing work [5, 7, 9, 15, 23, 28]: the streaming sources (each with data sorted in nonincreasing order of score) are read in a context-sensitive way according to their score upper bounds, and query processing ends once the k top-scoring answers are known. Both these aspects warrant an *adaptive* set of techniques that focus on reading the most promising tuples and executing all queries that use these tuples. To accomplish this, we use a novel coordinator called the **ATC**, which manages and routes tuples to different destination operators, using a *query plan graph* as a base instructor. In this section, we define the query plan graph and how it operates in more detail, starting with the operators and then discussing the ATC.

4.1 Query operators

The query plan graph represents operators as nodes and dataflows as edges. As discussed previously, the main node types include the *split* operator, the *multi-way join* operator (m-join), and the *rank-merge* operator. Refer to Figure 4, which shows a detailed excerpt of a query plan graph for our running example.

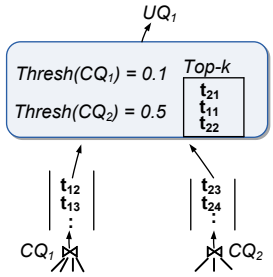

Figure 6: Query output streams and a rank merge operator.

Split operator (for subexpression sharing). Our example scenario shows how different conjunctive queries may have overlapping subexpressions whose computation might be performed once, then fed into multiple query plans. Figure 4 shows how the pipelined *split* operator is used to feed the output of a shared expression to multiple downstream m-joins or rank-merge operators.

M-way joins (for query computation). In producing ranked results, the choice of which input to read next is dependent on the relative scores of the inputs. A traditional binary join plan enforces a particular order of processing: $(R \bowtie S) \bowtie T$ will always produce RS subresults that get joined with T. A much more flexible scheme, which in different contexts is called the *m-join* [34] or the *STeM eddy* [24], is to generalize the pipelined hash join to support m-way joins. Here, each input has an associated *access module* [6][3] — against which other tuples may be probed to compute join results. As tuples are read from a streaming input, they are inserted into the access module, then probed against the other access modules according to a *probe sequence*. We also exploit the fact that this probe sequence can be adjusted at runtime based on monitored values for the various join selectivities, using techniques proposed in [24] — in particular, we monitor the number of tuples output from each probe into the hash table or access module, and update selectivities after a certain number of tuples have been read. This results in an adaptive form of optimization within the multi-way join that can use a *different* ordering for each input relation.

Rank-merge operator (for top-k). The results of a single user query must *merge* the outputs of multiple conjunctive queries, i.e., the outputs of multiple m-joins. The Q System generates query plans in a way that ensures the streaming sources are returned in nonincreasing order of score, and that the m-joins produce their results in nonincreasing order of score. We define an m-way *rank-merge* operator (Figure 6) that receives tuples from each query CQ_i, and uses each score function C_i to computes the threshold for the next value to be returned by CQ_i. It maintains a priority queue of the k highest scoring tuples seen from all conjunctive queries; from this, it outputs the highest-scoring tuple above all thresholds, and reads a tuple from the output stream that will drop the score threshold the most (generally, the stream with the highest threshold). This basic operation follows the the ideas of the Threshold Algorithm and No-random-access Algorithm of [7].

To illustrate the ideas above, we define $\mathcal{Q}$ as the current set of conjunctive queries given to the query batcher. We define an *input*

[3]For a streaming source, the access module is simply the associated hash table; for a random access source, the module is a wrapper for probing the remote site using the join key.

assignment as a pair ($\mathcal{I}$,**I**), where $\mathcal{I}$ is a set consisting of the logical subexpressions that will be computed outside the middleware (by the streaming and random access sources). The results of each input in $\mathcal{I}$ will be fed into (possibly multiple) queries in $\mathcal{Q}$. We use a map **I**: $\mathcal{I} \rightarrow$ P($\mathcal{Q}$), where P($\mathcal{Q}$) are all subsets of $\mathcal{Q}$, to relate the inputs to the corresponding queries: for any input J in $\mathcal{I}$, **I**[J] refers to the queries that will use J in their query plans.

EXAMPLE 4. *Refer to the query plan graph of Figure 4. The input assignment* $\mathcal{I} = \{\sigma(T), RL, G2G \bowtie GI, TS, TP_R, E2M_R, UP_R, I2G_R\}$ *with the subscripts by R denoting random access sources. The corresponding queries for input* $\sigma(T)$ *is* $\mathbf{I}[\sigma(T)] = \{CQ_1, CQ_2, CQ_3, CQ_4\}$. *Once a tuple from the stream of* $\sigma(T)$ *is read, it is split into two different paths that lead to two different m-join operators, and ultimately along paths that lead to join results for all "its" queries. There are probes into the hash tables created for* RL, $G2G \bowtie GI$ *and* TS, *as well as the random access sources of* $TP_R, E2M_R, UP_R$ *and* $I2G_R$. *Likewise, a tuple read from the streaming source* TS *is sent to the m-join operator that will create results for* CQ_1.

4.2 Coordinating Simultaneous Execution

Each rank-merge operator requests result tuples from its constituent conjunctive queries, which in turn indirectly request tuples from input streaming sources in $\mathcal{I}$. Given that different user queries have different score functions for their queries, their associated rank-merge operators may prioritize different input expressions. Yet the input expressions are often shared across queries, leading to contention.

To solve these issues, we exploit the fact that the m-join operator is in some sense *push-based*: given the arrival of a new tuple from any of its streaming sources, it stores and probes this tuple according to an optimized probe sequence. We can "holistically" choose an input to read, and ensure that its tuples are propagated through split operators to *all* conjunctive queries' m-join operators, which will pipeline the results downstream. The choice of *which* source should be read next — across all conjunctive queries — is dependent on the state of the thresholds in each rank-merge operator. The **ATC module** has the task of "looking across" the set of rank-merge operators' thresholds, and using this information to choose the next source to fetch from. We explored a variety of scheduling schemes, and found that a round-robin scheme worked best. Here we look at each rank-merge operator in every round, and we read from its preferred stream before moving on to the next query. This scheme has the same outcome as a voting strategy where the input stream with the highest number of tuple requests gets read the most. It also prevents starvation of sources, which might otherwise be neglected if only a few operators requested tuples from them.

5. QUERY PLAN GENERATION

We now consider how to choose the most efficient query plan graph for answering an *initial* batch of conjunctive queries $\mathcal{Q}$, forming one or more user queries. Section 6 discusses how we implement the dynamic behavior of the Q System over time.

The initial query optimization problem can be considered one of top-k multiple query optimization. Recall from our previous discussion that data originates from either a streaming or random access source. Our goal is to generate a query plan graph for the entire set of queries in $\mathcal{Q}$, which takes advantage of the query processing capabilities of the underlying remote data sources (if they support SQL), accommodates any restrictions on data access (via Web forms), and exploits overlap among the conjunctive queries.

Our system adopts a two-stage approach to generate a query plan graph. In the first stage, it does a cost-based exploration of possible strategies for *pushing query operations into the streaming data sources*: push-down can reduce the overall amount of computation to be performed *and* create common subexpressions used by multiple conjunctive queries. In the second stage, for the remainder of the query, the optimizer performs a heuristics-based *factorization* of the query plan graph into select-project-join subexpressions that are shared by multiple conjunctive queries, and generates a query plan graph with split, m-way join, and rank-merge operators. The next two sections describe each of these stages in turn.

5.1 Pushing Down Common Subexpressions

The optimizer's first task is to factor out from the set of queries an input assignment $\mathcal{I}$, to be executed at the remote DBMS sites. The results of each $J \in \mathcal{I}$ will be shared among multiple conjunctive queries, namely those queries in the set **I**[J].

Our goal is to find subexpressions large enough to source more than one conjunctive query, yet avoid forcing the optimizer to create a bad plan that requires streaming in too many tuples or executing expensive joins. This requires a form of multi-query optimization that is known to be intractable if done in exhaustive fashion: optimization of a single query already results in a number of plans exponential in the number of joins, and a full-blown dynamic programming enumeration of plans with shared subexpressions would compound this by the number of different ways of sharing subexpressions. Hence our optimizer, like other modern multi-query optimizers [35], employs a set of pruning heuristics in looking for opportunities to combine and push down subexpressions.

5.1.1 Pruning Heuristics

Consider queries as shared subexpressions. We estimate the overall cardinality for each query that can be pushed-down. If a query produces few results, we do not consider its subexpressions — unless those subexpressions are shared by a different (larger) set of conjunctive queries.

Only stream relations that have scoring attributes. Not all relations may have attributes contributing to the scoring function. For any relation R with no score-related attributes, if R is read as a stream, then generally it must be read entirely before a terminating condition can be reached (each tuple in R contributes equally to the score, so there is no change in the query threshold). Hence, we treat R as a source to be probed against, rather than to be streamed, *unless* its estimated cardinality is less than a threshold $\tau(R)$ [4].

Filter subexpressions by estimated utility. We consider subexpressions we deem to be "useful": those that are shared by a minimum number of conjunctive queries, or that have low cardinality. Conversely, if we know that certain subexpressions are expensive to compute at the source (for example, non key-key or key-foreign key joins), then these are pruned away. We always designate base relations from streaming sources as useful.

Do not consider overlapping pushed-down subexpressions. If a subexpression has not been pruned according to one of the heuristics above, then we consider it as a candidate only if for every query CQ, it is a subexpression of CQ, or it does not overlap with CQ. Intuitively, we would like to avoid overlapping subexpressions pushed down to the base sources, as this requires streaming in a base relation more than once.

We expect the set of subexpressions predicted to be useful to change over time, as better statistics are learned about base data

[4] $\tau(R)$ is set offline according to the relative cost of probing the source vs. streaming its tuples.

Algorithm 1 BestPlan($\mathcal{Q}$, ($\mathcal{S}$, **S**), ($\mathcal{A}$, **A**)):
$Input \rightarrow$ Query set $\mathcal{Q}$, Candidates ($\mathcal{S}$, **S**), partial inputs ($\mathcal{A}$, **A**)
$Output \rightarrow$ A plan P with the input assignment ($\mathcal{I}$, **I**), $\mathcal{A} \subseteq \mathcal{I}$

```
1: if ∃ cached plan P' for inputs A then
2:    return P
3: end if
4: if |S| = 0 then
5:    (I, I) ← (A, A)
6:    construct a plan P with inputs (I, I)
7:    C ← calculate cost of P
8:    return (P, C)
9: end if
10: (bestCost, bestPlan) ←dummy plan with cost ∞
11: for each input J in S do
12:    (S', S') ← (∅, ∅)
13:    for each input J' ≠ J in S do
14:       if J and J' share a common relation and
          S[J'] - S[J] is not empty then
15:          add J' to S' with S'[J'] = S[J'] - S[J]
16:       end if
17:    end for
18:    (A', A') ← (A ∪ {J} , A) ; A'[J] ← S[J]
19:    (P, C) ← BestPlan(Q, (S', S'), (A', A'))
20:    if C < bestCost then
21:       (bestPlan, bestCost) ←(P, C)
22:    end if
23: end for
24: cache bestPlan for A
25: return bestPlan
```

sources during dynamic operation (as described in the next section). The search process that we describe below is flexible enough to accept any combination of subexpressions and queries, and is guaranteed to find a *valid* combination of subexpressions (see below), so long as we include all base relations from streaming sources as useful subexpressions.

DEFINITION 1. *We consider the input assignment ($\mathcal{I}$, **I**) to be valid if for each query $CQ_i \in \mathcal{Q}$ and each relation R in CQ_i, there exists exactly one input $J \in \mathcal{I}$, such that J is a subexpression of CQ_i, involving R, and $CQ_i \in \mathbf{I}[J]$.*

5.1.2 Cost-Based Enumeration

We next enumerate the set of possible subexpressions satisfying the pruning heuristics. For efficiency, we employ a memoization structure called an AND-OR graph, commonly used in multi-query optimization [26]. The AND-OR representation of subexpressions is a directed acyclic graph that consists of alternating levels of two types of nodes: "OR" nodes that encode equivalent subexpressions, and "AND" nodes that encode selection and join operations.

The optimizer enumerates an AND-OR graph representing all subexpressions of all queries in $\mathcal{Q}$, except those pruned according to our heuristics above. We convert this set of subexpressions into a *candidate input assignment* ($\mathcal{S}$, **S**), where each candidate in $\mathcal{S}$ is an input expression to consider evaluating outside the ATC.

Given these candidates, our goal is now to find the input assignment ($\mathcal{I}$, **I**) such that $\mathcal{I} \subseteq \mathcal{S}$ for which the cost of computing all the queries in $\mathcal{Q}$ is minimum. Algorithm 1 shows our search procedure, which uses memoization and top-down search similar to the Volcano model [8]. Our input to the algorithm is a query set $\mathcal{Q}$, the candidates ($\mathcal{S}$, **S**) and and the input assignment ($\mathcal{A}$, **A**) that constitutes a partial plan. $\mathcal{A}$ is initially the empty set, and candidates from $\mathcal{S}$ are continually moved to $\mathcal{A}$ so that a valid input assignment can be found. At the end of each recursive step, $BestPlan$ returns a query plan P with inputs ($\mathcal{I}$, **I**) such that $\mathcal{A} \subseteq \mathcal{I}$ (i.e., P is the best plan that uses all inputs/subexpressions of $\mathcal{A}$).

After initially calling $BestPlan(\mathcal{Q}, (\mathcal{S}, \mathbf{S}), (\emptyset, \emptyset))$, we iterate over each candidate (lines 10-22) in $\mathcal{S}$ and find the best plan that uses that candidate along with others in $\mathcal{A}$. At each point $BestPlan$ chooses to use an expression J, it creates a modified candidate assignment ($\mathcal{S}'$,**S'**) in order to ensure queries using J will not also use overlapping expressions. Note that moving candidates from $\mathcal{S}$ to $\mathcal{A}$ ensures $\mathcal{S}$ decreases in size each time a recursive call is made. This, along with our step for adjusting the remaining candidates in $\mathcal{S}$ ensures that we will generate a valid plan.

PROPOSITION 1. *The plan P with input assignment ($\mathcal{I}$, **I**) returned by BestPlan is a valid plan for the query set $\mathcal{Q}$ and the set of streaming data sources. (Proof appears in the technical report [17].)*

EXAMPLE 5. *Consider our running example again when KQ_1 and KQ_2 are posed. The query set is $\mathcal{Q} = \{CQ_1, CQ_2, CQ_3, CQ_4\}$. In order to build the candidate input assignment ($\mathcal{S}$, **S**), the optimizer decides that a set of useful subexpressions [5] may be $G2G \bowtie GI$, $G2G \bowtie GI \bowtie T$ and $TP \bowtie E2M$, with $\mathbf{S}[G2G \bowtie GI] = \{CQ_1, CQ_2\}$, $\mathbf{S}[G2G \bowtie GI \bowtie T] = \{CQ_2\}$, and $\mathbf{S}[TP \bowtie E2M] = \{CQ_1\}$. After calling BestPlan($\mathcal{Q}$, ($\mathcal{S}$, **S**), ($\emptyset$, $\emptyset$)), the optimizer finds the best plan using each of these subexpressions. In order to find the best plan using $G2G \bowtie GI \bowtie T$, it will call BestPlan($\mathcal{Q}$, ($\mathcal{S}'$, **S'**), ($\mathcal{A}'$, **A'**)), where $\mathcal{A}' = \{G2G \bowtie GI \bowtie T\}$ and $\mathbf{A}'[G2G \bowtie GI \bowtie T] = \{CQ_2\}$; $\mathcal{S}' = \{G2G \bowtie GI, TP \bowtie E2M\}$ and $\mathbf{S}'[G2G \bowtie GI] = \{CQ_1\}$.*

5.2 Factorization of Query Plan Graph

Once the set of expressions to be pushed to the sources, $\mathcal{I}$, is determined by $BestPlan$, we are faced with the challenge of determining join and selection orderings within the middleware portion of the query plan graph P. Our approach is to first *factor* the query plan graph into different connected components with different amounts of sharing, and then to defer decisions about join ordering *within* each component to runtime, by computing all joins within that component using a single m-join that uses monitored selectivities to choose a sequence for probing. Our ultimate plan graph will look like the one of Figure 4, where each *split* or *rank-merge* represents the boundary between connected components.

To construct this query plan graph, we build up query expressions in iterative fashion. Start with a *frontier set* consisting of the streaming inputs in $\mathcal{I}$. For each E in this set, suppose E is a common subexpression for the set of conjunctive queries $\mathbf{I}[E]$.

- If possible, find the most selective join or selection operation o common to all queries in $\mathbf{I}[E]$, which can be applied to E. Replace E in the frontier set with $o(E)$ and iterate.
- Otherwise, insert a *split* operator over E, then choose a set of operators into which it feeds, using the following iterative process. Let $Rm = \mathbf{I}[E]$ and remove E from the frontier set. Choose the join or selection operation o_x such that $o_x(E)$ is common to the maximal number of queries in Rm (breaking ties by picking the most selective such operator). Add $o_x(E)$ to the frontier set. Now eliminate from Rm those queries that include $o_x(E)$, and repeat, until Rm is empty.

We repeat until the frontier set contains all expressions in $\mathcal{Q}$: we now have a complete query plan graph. Note that the problem presented here is another form of multi-query optimization — finding the best ordering of joins and selections in order to compute all queries, which is NP-hard in the number of joins involved. Instead, the greedy heuristic above is targeted at creating as few factored components as possible. This way, we defer as many decisions about operation ordering to the eddy modules. Changing orderings *between* factored components is an interesting challenge that for future work.

[5] We omit base relations for simplicity.

6. INCREMENTAL ATC MODIFICATION

We now discuss the *dynamic* aspects of our implementation — specifically, how state is maintained for reuse and removed to assist with newly arriving queries. When a batch of queries is first posed, these queries must be optimized given an existing query plan graph and its state; we discuss this in Section 6.1. To execute the resulting query, we must modify an existing query plan graph, and then notify the ATC about the new operators, as discussed in Section 6.2. Finally, under resource constraints we may need to discard some state from memory (Section 6.3).

6.1 Optimizing New Queries

As new queries arrive over time, they are likely to overlap with queries previously executed by the ATC. The optimizer must be able to determine whether it should reuse in-memory state (e.g., the hash tables within join operators) of prior query execution runs.

Updated cost estimates. The optimizer must be modified to properly estimate the costs of reusing existing tuples. After it receives a new batch of queries, the optimizer builds the new candidate assignment ($\mathcal{S}$, **S**) as in Section 5. For each subexpression $J \in \mathcal{S}$, if J represents a streaming source, it queries the QS manager about whether J is in memory (i.e., whether it has been used for a previous set of queries) and if so, the number of results that have been streamed in. Since the costing of plans is based on the number of tuples to be read from the source, the optimizer then adjusts the estimate of using J in a plan to account for any source tuples already read in. It next prevents J from being evicted, by requesting that the QS Manager "pin" J down. Finally, it runs the $BestPlan$ algorithm with the updated cost estimates.

Preventing over-sharing of results. One trade-off that may arise is due to the limited amount of concurrency in the ATC: in very large query plan graphs, any given query UQ_j may only depend on a small percentage of the overall graph. This can result in a phenomenon like thrashing in a multithreaded system: much of the ATC's resource budget gets spent on computing subresults for queries *other* than UQ_j. To improve concurrency, we can generate *multiple* query plan graphs, each with their own ATC. We accomplish this by *clustering* user queries in a simple hierarchical fashion. Given the initial set of conjunctive queries, we identify the most frequently occurring source relations in the workload. We build an initial cluster for each source by adding the set of user queries that reference the source, more than T_m times (where T_m is a configurable threshold). Then we repeatedly merge clusters whose Jaccard similarity (i.e., ratio of overlap) exceeds a second threshold T_c, until it is no longer possible to merge. The resulting set of clusters defines the sets of user queries to separately group, optimize, and execute.

6.2 Grafting New Query Plan Graphs

The optimizer passes a new query plan graph to the QS manager, to *graft* it onto the existing query plan graph. This is done by individually merging each path in the new query plan graph, as sketched below. For each node n in a path, iterating from source (leaf) to rank-merge operator, assign a *mapping* $m(n)$ from n to a query operator as follows.

1. If a query operator in the existing query plan matches n, let $m(n)$ be that operator.
2. If n's parent operator n' matches $m(n)$'s parent operator, then repeat the above step for n'.
3. Stop at any node n whose parent node n_p has no match in the existing query plan graph, then (if necessary) inject a split operator between $m(n)$ and its parent in the existing graph. Create a new operator for n_p, let $m(n_p)$ be this operator, and connect one of the outputs of the split operator to $m(n_p)$.

For any unmatched node n in the new graph, create a new operator for $m(n)$.

In order to prevent race conditions, we create the new query plan graph segments first, then suspend execution of the original graph and lock it from concurrent modification. Next we ensure that for each n in the new graph and its parent n_p, ensure that $m(n)$ feeds data to $m(n_p)$. Finally, the new conjunctive queries are registered with downstream rank-merge operators and the main ATC, so that thresholds are maintained. Whenever a new query reuses an old stream, the threshold is initialized to the current score upper bound of that stream. Execution may now resume.

EXAMPLE 6. *Consider again, our running example, when KQ_3 is posed (some time after KQ_1 and KQ_2). The optimizer decides that a plan for these queries is $\mathcal{I}$ = {$G2G \bowtie GI$, TS, T}, with* **I**[$G2G \bowtie GI$] = {CQ_5, CQ_6}, **I**[TS] = {CQ_6}, **I**[T] = {CQ_5, CQ_6}. *As the first step, a new rank merge operator is created for UQ_3. Then, the new plan graph created for CQ_5 and CQ_6 is merged with the original graph in Figure 4. We show a part of the merging process for the inputs T and TS in Figure 5. Observe two new split operators are added in order to route intermediate tuples of $\sigma(T)$ and $\sigma(T) \bowtie (G2G \bowtie GI)$ to new destination operators.*

Reusing state. A major complexity is that a new conjunctive query CQ_i may make use of data from input streams that have already been read. In such an event, simply reading further from the streams is insufficient; we must first re-process the earlier parts of the streams, which are buffered within the query plan graph's state.

A naive approach is simply to find the various streams' tuples buffered in the hash tables of existing m-joins, and to simply iterate over these tuples and join them. Not only would this require us to suspend normal query processing until the new join was complete — but in fact such an approach is score-unaware and would emit tuples in a way that do not conform to the downstream rank-merge operator's requirement that each conjunctive query's tuples be generated in nonincreasing score order.

The problem is that we need to re-process the tuples according to their original order; we accomplish this with a more sophisticated approach. We modify the hash tables to also embed a *linked list*: each time we add a tuple to the hash table, we update the *last*-added tuple to include a *next* pointer to the newly added tuple. Given the initial pointer to the first tuple added, we can visit the set of tuples in the order they were received from the input stream. Every time the QS manager provides a new set of queries to the ATC, it increments a counter called the *epoch* (a logical timestamp). Then, for each CQ_i that references input streams that have already been read, we create an additional new query CQ_i^e, to compute all the missing tuples for CQ_i. This query takes as its inputs the contents of the appropriate linked lists as recorded *before* epoch e, in order to avoid the introduction of duplicate results. Internally, we keep track of the association between tuples and epochs by storing each epoch's tuples in a separate *partition* within the m-join hash tables. Note that once CQ_i^e's plan is created, it is just another conjunctive query, and the linked lists from the hash tables are treated as another input in $\mathcal{I}$. The new query is registered with the rank-merge operator which takes into account the upper bounds for the linked lists just as another ranked input, in order to maintain correct thresholds.

Algorithm 2 shows the steps to recover missing tuples: RecoverState is called when the QS Manager discovers that the streaming sources for CQ_i have been read before its arrival. Lines 2-5 show steps to partition the hash table for each streaming input J that be-

Algorithm 2 RecoverState(CQ_i, ($\mathcal{I}$,**I**)):
Input → Conjunctive query CQ_i, Current assignment ($\mathcal{I}$, **I**)

```
1: e ← current epoch
2: for each input J ∈ I s.t. J is a stream and CQ_i ∈ I[J] do
3:     create a new hash table H_J^e where new tuples will be indexed
4: end for
5: CQ_i^e ← CQ_i, C_i^e ← C_i
6: Choose a streaming input J s.t. CQ_i ∈ I[J]
7: Let J_e be a streaming source which is the linked list of tuples from
   hash tables H_J^l where l < e
8: add J_e as a streaming source to I with I[J_e] = {CQ_i^e}
9: for each J' ∈ I s.t. J' ≠ J and CQ_i ∈ I[J'] do
10:    if J' is a streaming input then
11:        add J'_e as a random access source to I with I[J'_e] = {CQ_i^e}
12:    else
13:        I[J'] = I[J'] ∪ {CQ_i^e}
14:    end if
15: end for
16: Add the new query CQ_i^e with updated assignment (I,I)
```

longs to $\mathbf{I}[J]$. Note that the tuples that arrived before epoch e are exactly those tuples that need to be joined for recovering the missing tuples of CQ_i — these results are represented by the query CQ_i^e which is created in line 5. Now, since CQ_i^e requires a set of inputs, we reuse exactly the set of inputs for CQ_i, but we choose one streaming input J that will be the streaming source for CQ_i^e, (lines 6-8). Every other input J' is treated as a random access source (since tuples from J'_e are already indexed in a hash table). Lines 9-15 add the new query and modify the input assignment to $\mathcal{I}$.

6.3 Query Termination and Discarding State

Once a conjunctive query has completed, or can longer contribute to top-k output (its threshold is lower than the k^{th} tuple in the ranking queue of the downstream rank-merge operator), it gets unlinked from the query plan graph and deactivated. We simply traverse the graph *backwards* starting from the query node, removing nodes and edges until we reach a split operator. The split operator can only be removed if it does not route tuples along another path. If the split gets removed, we repeat the removal process recursively.

A natural question is when we remove the state for a query that is terminated. In a conventional OLAP query processing setting, one would expect that our basic approach — keep everything around in memory as a large cache for re-use — may be impractical due to large intermediate result sizes. In our setting, we expect memory and resource problems to be very infrequent on a modern machine, as ranking queries tend to read only (quite small) prefixes of relations, and most machines have multiple GB of RAM. Nonetheless, for completeness we develop strategies for cache replacement. Two types of objects are considered "cacheable": the contents of ranking queues that hold pending tuples to be output to the user, and hash tables corresponding to specific query subexpressions. Such items can be fully evicted if unreferenced by running or pending queries; or flushed to disk otherwise.

We experimented with a variety of potential cache replacement policies. The most obvious policy is least-recently-used (LRU), but we also considered factors such as result size and recomputation cost. We found that LRU, with size as a tie-breaker, worked quite well in practice. Since the results were not particularly informative, we omit them from the experiments section.

7. EXPERIMENTAL ANALYSIS

Our Q System implementation comprises approximately 50,000 lines of Java code and runs as a middleware layer over remote SQL databases. Evaluation was done using Java 6 JDK 1.60_04 on a dual-processor, dual-core Xeon 5140 machine with 8GB RAM and Windows Server 2003, using JDBC to connect to MySQL databases running on an identically configured machine. We briefly describe our experimental setup and methodology.

UQ	1	2	3	4	5	6	7	8
Queries	12.75	4.5	11	13.5	8.5	6	13.75	8.75

UQ	9	10	11	12	13	14	15
Queries	3.75	5.75	13.75	3.25	12.5	13	7.75

Table 4: Average number of conjunctive queries executed to return top-50 results over synthetic datasets

To create conjunctive queries from keyword searches, we used the Q System's query generator algorithm of [33]. We used both synthetic and real-world data sets.

Synthetic workload. Our synthetic dataset made use of the Genomics Unified Schema [21] (GUS), which has 358 relations. We created 4 simulated database instances by populating the relations in schema with 20,000-100,000 randomly generated tuples apiece. Each synonym/relationship table in the schemas was extended with an additional attribute representing a similarity score. Scores, join keys, and coefficients on the score functions for the various user queries were drawn from a Zipfian distribution. We stored these instances in MySQL, indexed by join keys and score attributes.

We generated a suite of 15 user queries by choosing pairs of keywords from a list of common biological terms, using a Zipf distribution on the keywords[6]. Each such query yielded a maximum of 20 conjunctive queries over GUS. For each relation matching a search term, we also added a synthetic score attribute, simulating an IR-style keyword similarity score.

Real-data workload. We also conducted tests over real data from the biological datasets Pfam (`http://pfam.sanger.ac.uk/`), a collection of protein families with multiple relationship tables to protein sequences, and Interpro (`http://www.ebi.ac.uk/interpro/`), another integrated database of protein families and sequence information. The former database contains a mapping table that relates Pfam families to Interpro entries. We created 15 keyword queries using the same methodology as in our synthetic case, using keywords that matched to sequence, family, and publication data.

Delays. To simulate wide-area delays over our local-area network, we added random delays for each tuple read from a data stream and each join probe performed against a remote DBMS. Delays were chosen from a Poisson distribution with an average of 2 milliseconds. Keyword search queries were posed within 6 seconds of one another. All graphs include averaged values taken over three different runs over each database instance (12 runs in total), and we include 95% confidence intervals.

Overview of experiments. Our experiments seek to answer the following questions: What are the performance implications of (1) shared subexpressions and (2) state reuse? (3) Given state reuse, is shared subexpression computation even necessary? (4) How does the time to perform multiple query optimization affect overall performance? (5) Do our results, done over synthetic data, generalize to real-world data?

7.1 Benefits of Sharing

We first study the impact of sharing and result reuse. Each user query in our query workload gets expanded into a set of conjunctive queries: refer to Table 4 to see how many conjunctive queries were

[6]Keyword queries can be found in `https://dbappserv.cis.upenn.edu/home/?q=node/132`

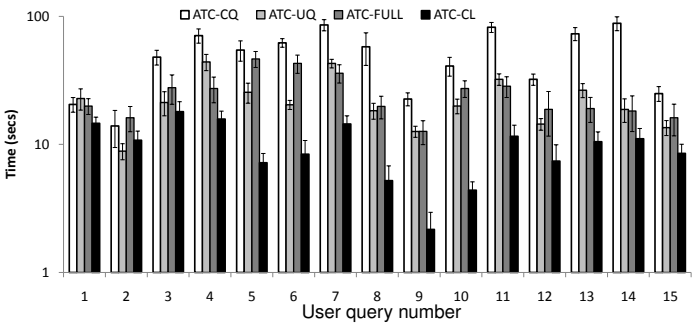

Figure 7: Running times to return the top-50 results for each user query.

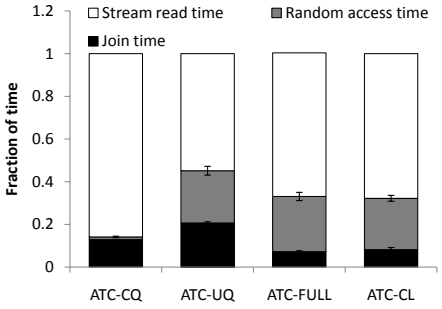

Figure 8: Breakdown of execution time.

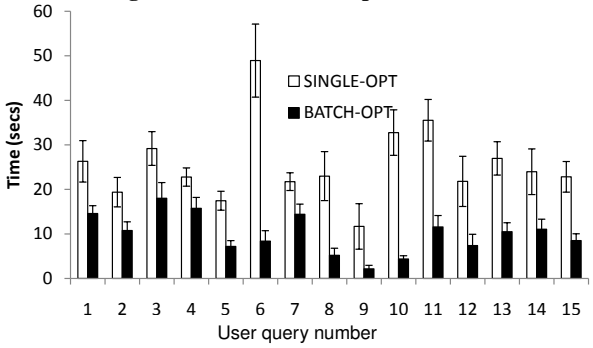

Figure 9: Running times: individually versus batch-optimized queries.

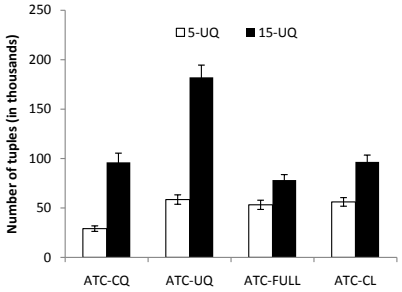

Figure 10: Total work done, 5 vs. 15 UQs

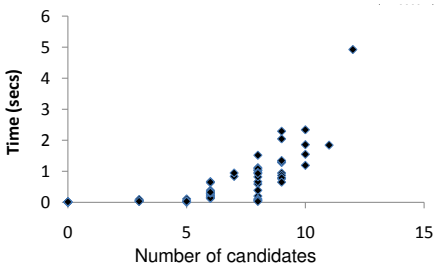

Figure 11: Optimization times vs. candidate inputs.

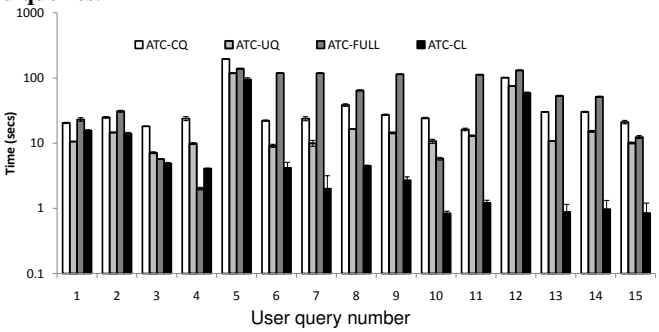

Figure 12: Execution times over the Pfam/Interpro dataset.

required to return the top-50 results for each user query, averaged across the four different synthetic data sets. Recall that the QS manager and ATC coordinate in order to ensure that additional CQs are executed only as necessary to return relevant results. In our experiments, we never needed more than 20 CQs per user query.

We studied several different optimization and QS manager configurations:

- As a baseline, we *separately* optimize each user query and *disable sharing* across conjunctive subexpressions; we refer to this method as **ATC-CQ** since the ATC executes conjunctive queries as separate m-joins whose results are merged.
- We enable subexpression sharing *within* a given user query, but disable it *across* queries. We refer to this as **ATC-UQ**.
- We use and update a *single* query plan graph to execute all user queries received at any point in time; this process automatically reuses existing state. This is strategy **ATC-FULL**.
- We manually clustered the user queries to create *multiple* query plan graphs when different sets of queries had significantly different inputs (see Section 6.1 for a way of doing this automatically). We call this **ATC-CL**.

In all configurations, the sequence of queries is user posed over time, with random delays of up to 6 seconds. Our query batcher is set to group queries into batches of size 5. Figure 7 shows the resulting execution times for each user query. We see that virtually across the board, sharing within a user query (**ATC-UQ**) provides benefits versus the baseline (**ATC-CQ**). When we look at sharing *across* user queries, we see that a different issue arises. **ATC-FULL** only performs better than **ATC-UQ** in 5 out of the 15 queries: here, several different rank-merge operators are requesting tuples from different subexpressions, seeking to optimize a different goal; the resulting contention means each query suffers a delay waiting for the others. We see that the clustering method, **ATC-CL**, handles this problem well by separating the contending queries and increasing parallelism.

Overall, we see that subexpression sharing and reuse both provide significant benefits — up to 90% for Query 9. We consider whether the two kinds of sharing can be used in isolation in later experiments, but first look in more detail at a breakdown of time versus query execution operations (Figure 8). Here we divide the total time into three operations: reading tuples from the streaming sources (*Stream read time*), performing in-memory joins (*Join time*), and probing remote sources with tuples to perform a two-way semijoin (*Random access time*). **ATC-UQ**, **ATC-FULL** and **ATC-CL** spend much less time reading tuples from the base streams than **ATC-CQ**, as they optimize by sharing and reusing tuples. On the other hand, they spend much more time probing against remote

sources: we attribute this to the fact that probes are commonly used against relation sources that have no scoring attributes, and that the ATC may need to execute a fairly large number of probes in order to maintain the full set of results required to properly maintain the query threshold. Given that we cache tuples from random probes, we can expect the rate of probing to decrease over time.

7.2 Benefits of Subexpression Sharing

We next consider whether the process of identifying shared subexpressions — i.e., multiple query optimization — makes a useful contribution, given that in principle reuse of results from one run to the next might achieve similar behavior. To study this, we took the **ATC-CL** configuration and our query workload, and compared their running times when each query was optimized separately (batch size = 1) versus multiply (batch size = 5, as in the previous experiments). Figure 9 shows significant gains in performance for larger batch sizes, clearly indicating that it is advantageous to proactively identify opportunities for subexpression sharing.

7.3 State Reuse across Time

Assuming adequate memory and sufficient overlap among queries in the workload, we would expect that the *incremental* cost of answering newly posed queries should go down over time, if we can reuse subexpressions. Figure 10 shows that this is indeed the case for our query workload: it plots the total amount of work done (measured as the total number of input tuples consumed) in answering the first 5 queries of the workload, versus the full set of 15 queries. We see for the cases where tuples are not reused (**ATC-CQ** and **ATC-UQ**) that executing 15 queries takes approximately three times the amount of work as executing 5 queries. On the other hand, for **ATC-FULL** the complete suite requires only about 75% more work, and for **ATC-CL** about twice the amount of work. (Recall that **ATC-CL** shares less than **ATC-FULL**, which reduces contention and thus running time, but actually does slightly more work as measured by intermediate result production.)

7.4 Optimizer Running Times

While our previous timings included query optimization as a component, we now focus on the actual cost of performing multiple query optimization. In general multiple query optimization costs are determined by the number of unique query atoms (source relations), and in this case the main portion of the search is the number of candidate expressions considered for push-down into the source relations. We measured the amount of time spent optimizing the 15 user queries, batched in groups of 5. Figure 11 shows the results for a single run (other runs showed similar trends), where we plot the number of candidate subexpressions for a set of queries, against the time taken to generate a plan. Not surprisingly, the distribution follows an exponential curve as the number of candidates increase.

7.5 Generalization to Real Data

Finally, we investigated whether our conclusions transfer to queries over real rather than synthetic datasets. Given our combined Pfam-Interpro dataset, we used MySQL's text search capability to match pairs of two-keyword phrases against database tuples, capturing MySQL's similarity score as well as the tuples. Additionally, we included one additional score attribute, namely the age (year) of publication. Each user query here resulted in 4 conjunctive queries; we posed the user queries in sequence with random delays of up to 6 seconds. Figure 12 shows the execution times of all queries using our various configurations, for $k = 50$ results.

As with the synthetic data, **ATC-UQ** tends to provide a minor improvement over **ATC-CQ** (with a best case of 77% better for UQ_5). **ATC-FULL**, on the other hand, showed few gains — this was because the real dataset resulted in significantly larger amounts of data, hence more computation time and more contention delay in our middleware layer. The **ATC-CL** confugration clustered the user queries into three query plan graphs (queries 3 and 12 in one plan graph; query 5 in another; and the remaining queries in the final graph). This less-contentious arrangement provided significant improvement, especially in queries 7 through 15, with a maximum performance gain of 97% over **ATC-CQ** and 90% over **ATC-UQ**. All told, the results over real data are very consistent with those over synthetic data.

Our overall conclusion from the experiments is that our techniques for subexpression sharing, and for threshold-based selection of input tuples via the ATC, provide significant performance benefits. Our studies show that clustering user queries together can also alleviate the problem of contention occurring due to a high rate of incoming user queries. We plan to further investigate techniques for eliminating contention and I/O bottlenecks, including selective use of multithreading over portions of the same query plan graph.

8. RELATED WORK

This paper addresses a distributed, data integration-centric application of keyword search over databases [2, 11, 13, 18, 27, 33], with concurrent and ongoing sequences of user query requests. Our approach adopts a fairly general model and ranking scheme; it can be directly incorporated into any of the existing systems that generate queries before executing them [11, 13, 27]. Some other systems like [2, 11, 18] tend to interleave query generation with tuple retrieval and intermediate result creation, and there our techniques for performing multiple query optimization are somewhat more difficult to fit architecturally.

Our goal, through the ATC and query plan graph, is to support efficient top-k query answering [9, 23], but across multiple queries simultaneously (and allowing for reuse). The problem of developing efficient algorithms for executing these queries includes the Fagin et al. threshold and no-random-access algorithms [7] and work on (pairwise) rank joins [15, 28]. Our multiway join is essentially a STeM eddy [24], which uses adaptivity to determine an order of join evaluation. It also resembles the m-join [34], with the distinction that the control logic is separated into the ATC, rather than into the operator.

The problem of optimizing individual top-k queries was considered in [16, 29], and we leverage their cost estimation techniques. So far as we know, multiple query optimization has only been studied in the unranked context [26, 30, 35]. The related problem of view selection and materialization for ranked queries, as well as merging of ranked union queries, is studied in [14].

In many ways, our work resembles the continuous query execution model considered in NiagaraCQ [4] and other related systems, where the goal was to optimize an initial set of queries and produce a continuous query plan; then to leverage the existing plans for executing subsequent queries. Our context is quite different, in that the queries are not truly continuous, so certain query subexpressions truly become irrelevant and may be evicted; and the top-k setting changes the query execution mode.

9. CONCLUSIONS AND FUTURE WORK

This paper has addressed the issue of improving keyword search performance for data-integrating queries, through sharing and reusing subexpressions, as well as adaptive query processing techniques. Our contributions were:

- A new top-k query processing architecture that keeps in-

memory state and existing query plan structures from one query execution to the next, enabling effective result reuse and recomputation over time.

- An optimizer that combines cost-based and heuristic search, in order to find common subexpressions across and within top-k queries, and which supports the reuse of existing subexpressions from past query executions.
- A fully pipelined, *adaptive* top-k query execution scheme for answering multiple queries, consisting of a query plan graph of m-joins (STeM eddies) and rank-merge operators, supervised by a novel ATC controller.
- A query state manager that can *graft* and *prune* elements in an executing query plan, and which manages the reuse and eviction of state.
- A comprehensive set of experiments over synthetic and real datasets, demonstrating the performance gains of our approach.

As future work, we plan to study opportunities to strategically exploit threads to limit contention among different queries; to extend our query processor to consider more complex queries that include aggregation; and to consider the problem of incremental maintenance of views defined over multiple queries.

Acknowledgments

This work was funded in part by NSF grants IIS-0447972, 0713267, and 1050448, and DARPA grant HRO1107-1-0029. We also thank the reviewers for their helpful feedback.

References

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *ICDE*, 2002.

[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[3] H. Cao, D. Jiang, J. Pei, Q. He, Z. Liao, E. Chen, and H. Li. Context-aware query suggestion by mining click-through and session data. In *KDD*, New York, NY, USA, 2008.

[4] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *SIGMOD*, 2000.

[5] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD*, 1998.

[6] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 2007.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4), June 2003.

[8] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *ICDE*, 1993.

[9] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an RDBMS for web data integration. In *WWW*, 2003.

[10] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

[11] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[12] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.

[13] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[14] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *The VLDB Journal*, 13(1), 2004.

[15] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, 2003.

[16] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, H. G. Elmongui, R. Shah, and J. S. Vitter. Adaptive rank-aware query optimization in relational databases. *ACM Trans. Database Syst.*, 31(4), 2006.

[17] M. Jacob and Z. Ives. Sharing work in keyword search over databases. Available from `www.cis.upenn.edu/~majacob/papers/techATC.pdf`.

[18] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[19] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, 2008.

[20] H. Kautz, B. Selman, and M. Shah. Referral web: combining social networks and collaborative filtering. *Commun. ACM*, 40(3), 1997.

[21] J. C. Kissinger, B. P. Brunk, J. Crabtree, M. J. Fraunholz, B. Gajria, A. J. Milgram, D. S. Pearson, J. Schug, A. Bahl, S. J. Diskin, H. Ginsburg, G. R. Grant, D. Gupta, P. Labo, L. Li, M. D. Mailman, S. K. McWeeney, P. Whetzel, C. J. Stoeckert, Jr, and D. S. Roos. The Plasmodium genome database: Designing and mining a eukaryotic genomics resource. *Nature*, 419, 2002.

[22] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, 2005.

[23] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2), 2004.

[24] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[25] N. Roussopoulos and H. Kang. A pipeline [sic] n-way join algorithm based on the 2-way semijoin program. *IEEE Trans. on Knowl. and Data Eng.*, 3(4), 1991.

[26] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, volume 29(2), 2000.

[27] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational databases. In *ICDE*, 2007.

[28] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, 2008.

[29] K. Schnaitter, J. Spiegel, and N. Polyzotis. Depth estimation for ranking query optimization. In *VLDB*, 2007.

[30] T. K. Sellis. Multiple-query optimization. *TODS*, 13(1), 1988.

[31] X. Shen, B. Tan, and C. Zhai. Context-sensitive information retrieval using implicit feedback. In *SIGIR*, New York, NY, USA, 2005.

[32] P. P. Talukdar, Z. G. Ives, and F. Pereira. Automatically incorporating new sources in keyword search-based data integration. In *SIGMOD*, 2010.

[33] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. In *VLDB*, 2008.

[34] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.

[35] J. Zhou, P.-Å. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.