FluxQuery: An Execution Framework for Highly Interactive Query Workloads

Roe Ebenstein Niranjan Kamat Arnab Nandi
Computer Science & Engineering
The Ohio State University
{ebenstei,kamatn,arnab}@cse.osu.edu

ABSTRACT

Modern computing devices and user interfaces have necessitated highly interactive querying. Some of these interfaces issue a large number of dynamically changing and continuous queries to the backend. In others, users expect to inspect results *during* the query formulation process, in order to guide or help them towards specifying a full-fledged query. Thus, users end up issuing a fast-changing workload to the underlying database. In such situations, the user's query intent can be thought of as being *in flux*.

In this paper, we show that the traditional query execution engines are not well-suited for this new class of highly interactive workloads. We propose a novel model to interpret the variability of likely queries in a workload. We implemented a cyclic scan-based approach to process queries from such workloads in an efficient and practical manner while reducing the overall system load. We evaluate and compare our methods with traditional systems and demonstrate the scalability of our approach, enabling thousands of queries to run simultaneously within interactive response times given low memory and CPU requirements.

1. INTRODUCTION

Interactive experiences using modern devices such as netbooks, iPads, smart watches, Google Glass, etc. have become increasingly popular. Constructing a query interactively is common whether it be autocompletion [27], instant search for search engines, using gestures on iPads and touch-based devices [26, 35], or through image analysis in augmented reality interfaces [43, 48]. In such cases, the user's intent (and the final query) is not explicitly clear to the system *during* the construction of the query. The query input terminates when the user finalizes her actions and is happy with the constructed query. Current systems typically do not aid the user during this query construction or specification process; the user does not get result feedback for each exploratory query formulation. This makes it hard for the user to verify the impact of each query clause, or if the constructed query yields the *intended* query results. User studies have found that this type of query construction occurs over query specification sessions that take in the order of tens of seconds [35], and that *interactive-level* latency necessitates response to the user within the sub-second range [8, 10, 19, 31, 34, 42]. Thus,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3531-7/16/06...\$15.00

DOI: <http://dx.doi.org/10.1145/2882903.2882945>

the interface can provide a preview of the results *while* the query is being formulated, thereby, assisting to formulate the final query. Since the traditional database paradigm is that of “query→result”, the concept of interactive-level feedback *during* the construction of a query is challenging. As shown in the following sections, it is difficult to handle these tasks at the application layer, and thus requires rethinking of the underlying database design.

1.1 Interactivity & Ambiguous Query Intent

Often, the user is not completely aware of the data, the schema, or the query language. This presents a challenge to the user *during* the process of specifying her *query intent* – the process in which the system finds the user-intended query out of all the possible queries, going through a process which can be simply given as “ambiguous intent → unambiguous query”. Once an articulated query exists, the query can be executed – “query→result”. While decades of work exist in database systems towards query processing, the challenge of aiding a user to articulate her intent is comparatively novel.

Providing the user with an instantaneous/interactive feedback allows her to understand the result space better, which in turn aids the query specification process. For a complex database, the space of possible queries is large. Executing all these simultaneously, while considering large data size, is often expensive, and thus hinders exploratory querying abilities. Clients have limited resources, therefore a solution is needed at the engine level. As highly interactive workloads become increasingly common, this need intensifies.

1.2 Applications

Consider the case of performing an exploratory equijoin using hand gestures on a visual query interface running on a multitouch touch screen, a well-motivated scenario detailed in [35]. Here, while the system helps the user in specifying a query, it is unclear which tables and columns the user wishes to join until the user approves of the system's recognition. A preview of the query results enables the user to express her intent faster, allowing her to continuously skim resultsets of multiple queries rapidly – a use case which requires queries to continue execution after each preview is stopped being viewed. In addition, once the intended query has been identified, a part of the result is already available for processing, and the rest can be continuously processed. We focus on simple equijoins for the illustrative examples, the system works similarly for other types of joins (including outer joins) – the join criteria is orthogonal to the methods presented here.

Our exploratory join scenario is quite common for the query formulation process – real-world databases sometimes lack the documentation or primary / foreign key relationships explicitly annotated on schema. Let the two tables to be joined be denoted by T_1 and T_2 . Thus, the queries at hand are denoted by:

$$\tilde{Q}_{gesture} = T_1 \bowtie T_2 \quad (1)$$

where the conditions on the JOIN are determined by the user. During the gestural query articulation process, as the user drags columns of each of the tables close to each other by using a touch screen, the system is not certain of what the final user query would be, and shows a preview of the results for the most likely intended queries. Assuming the two tables T_1 and T_2 have c_1 and c_2 columns respectively, in the worst case, there can be $c_1 \times c_2$ different joins possible, and hence $\bowtie$ is used to denote that the join is *in flux*. Although not all join options have high likelihood, in an expansive schema, the number of queries can be big enough to overload current systems. We have experienced that systems become overloaded once the number of queries becomes greater than twice the number of cores. Since touch screens encourage a high level of interactivity, the rate of exploration increases – the user can consider each of the combinations in a rapid and arbitrary manner resulting in a rapid fluctuation of the probabilities of each query to be the user-intended one. Further exacerbating the situation is that tables might be huge, resulting in each join taking a long time to complete. As the user explores through all possible combinations, she settles on one of the JOIN combination as her final intended query.

This example is merely an illustrative example of a highly interactive application, and not the only use case. A similar use case can be found in web exploratory searches [36]. In addition, visual analytics tools, scientific data exploration tools, etc. are moving towards providing highly interactive experiences, which we expect will result in similar workloads.

Furthermore, orthogonal research, such as query expansions [32] and query recommendations [20], requires database engines to respond well to in-flux queries, and assume they do. Both produce in-flux queries, and the engines can use the concepts presented here to decrease the dependence of the techniques on the limitations of the underlying databases.

1.3 Challenges

Interactive querying provides us with numerous challenges. As shown in Tables 4-6 in the evaluation section, current databases are unable to provide interactive response time under heavy workloads.

- **Number of Queries:** The number of possible queries can be large. Even in the case of a single column equijoin between 2 tables T_1 and T_2 having c_1 and c_2 columns respectively, the maximum number of queries equals $c_1 \times c_2$. Processing even a relatively small fraction of these can get unwieldy for large tables. The user expects interactive response times and executing each query as the user transitions between them would overwhelm most existing systems.
- **Query Variability:** The number of different queries produced (or *churn*) in an interactive setting can be large. During the query specification process, the space of possible queries is initially large, and includes all the $c_1 \times c_2$ join options. The user may consider numerous queries before finalizing her intent, decreasing the space to a single query at the end. Thus, the rate of issuing (and canceling) queries poses a serious challenge.
- **Response Time vs. Throughput:** While increasing the throughput of the joins has been researched thoroughly, decreasing the response time has not been studied at the same depth. Current research primarily focuses on decreasing the overall query execution time, at the expense of responsiveness. Interactivity requires a short response time, while throughput is a secondary concern.
- **Highly Interactive Querying:** Interactive querying often refers to short response time, in the order of seconds or at the most minutes. Highly Interactive Querying, on the other hand, requires: 1. response time within milliseconds at the most (here, we use 500ms as a binary interactivity threshold); 2. responsiveness -

the system should provide immediate feedback for each user request; 3. adequate number of results for the user to explore (we use 500 results as the threshold). Current systems are not intended to be used within such constraints, and as shown later often do not respond well.

1.4 Contributions and Outline

We make the following contributions:

- We introduce a novel *query intent* model that allows us to properly represent highly interactive query workloads (Section 2).
- We describe *FluxQuery*, our framework for interactive query execution, based on a novel cyclic join approach (Sections 3-4).
- We provide algorithms for scheduling and executing highly interactive query workloads (Section 4), including novel join algorithms – the *FluxJoin* and the *Fast FluxJoin*.
- We experimentally demonstrate that unlike traditional databases, our implementation can successfully handle highly interactive query workloads within interactive response times (Section 5).

2. THE QUERY INTENT MODEL

The set of valid queries that can be issued to the database at any given time is considered as the **intent space**. Thus, the **query intent** is defined as the likelihood of each query in that space, represented by a probability distribution function (PDF) over all possible queries in the intent space. The determination of the query PDF is external to our system – we do not control it, but need to satisfy all the query constraints it demands. Without any input, the PDF is uniform – all queries are equally likely. At the end of the query specification, all but one of the queries are ruled out (i.e. probability set to 0) and the probability of the unambiguous intended query is set to 1.0.

During the user's interaction (such as the touch-based dragging gesture), the user starts with a perfectly ambiguous query and each action results in a change in the likelihood of the queries. We can formalize this concept by

$$\tilde{Q} = \{PDF_{t=1..TT}(Q_{1..q(t)}(T_{1..k}))\} \quad (2)$$

where $\tilde{Q}$, the **query intent transition**, is the set of all speculative queries and their probabilities during the lifetime of the specification of the query; $T_{1..k}$ are the tables on which the interactive query acts; and $t = 1..TT$ are the points in time through the query intent transition. The most difficult scenario to handle is if all queries change, e.g. all queries in $t = n$ do not appear in $t = n + 1$, etc. FluxQuery is designed to limit the effect such a churn has on the system. The PDF holds the first order Markovian property, i.e. the PDF at time $t + 1$ depends only on the PDF at time t and the user interaction – the function of change in the query likelihoods is smooth, with the gradient and current state determining the next.

Throughout this paper, we will consider and demonstrate using a simple single-column equijoin to be the running example, since it is the most challenging and critical component of highly interactive querying. However, as we will see the model can be extended to most of the query types.

In Figure 1, we demonstrate the *trembling* effect, similar to the one discussed in [28]. It demonstrates the likelihood of a query to be the intended one over time. Deciding the user-intended query becomes challenging because of the observed fluctuations. The consequences of stopping query execution and restarting it later are severe; therefore, we accommodate execution strategies which helps us in deciding when to start, stop, and re-start queries.

Figure 1: Query intent transition comprising 4 queries, during an interactive query specification task. All possible queries are equally likely at the start of the query specification, and 1 query emerges as the unambiguous query at the end.

2.1 Execution Strategies

Given that the query intent transition represents rapidly changing likelihoods of possible *queries*, the goal of the database is to provide useful results for all user queries during this process, pruning query options that are not probable while executing the rest. Considering the *bursty* and rapid changing of query probabilities, we list two execution strategies for deciding which queries to continuously execute and which to stop.

Full Query List Execution: In cases where all likely queries need to be executed, a tremendous load is put on the database. We consider this as *full execution*, where the database engine has to execute all the queries in the query intent space, i.e., all nonzero probability queries at any given time. We note that this workload refers to the queries we execute, not to the query results. We assume that all query results produced by the join execution while the query is registered are needed. FluxQuery is allowed to remove a query from the query execution list only if it completed its execution – the query list is client-controlled.

Query List Top- k Execution: In applications such as the multi-touch gestural join, due to the *pdf* Markovian property, it is possible to save some computing resources by using the *top- k* strategy. Since only the most probable queries are likely to be previewed to the user, there is lesser value in executing unlikely queries, other than because they might become likely later. Selecting the right k is a difficult task – a low k value would result in bursts of queries being sent to the engine, while a high value of k would behave like the full execution approach and is an interesting topic for future research. In this scenario, FluxQuery is allowed to stop and restart queries based on likelihood. We note that the top- k refers to the k queries with the highest likelihood, and not to the top- k results.

3. THE FLUXQUERY FRAMEWORK

3.1 System Architecture

We propose a centralized main-memory execution engine based on the idea of continual *circular clock scans* [4, 5, 12, 52], where concurrent queries register to ongoing scans, much like passengers getting on and off a cyclic elevator. This family of scan strategies allows reduction in latency, and prioritization of buffer page loads, and is ideal for **interactive query execution** based on the workloads presented in Section 2. The most likely subset of queries the user might intend to execute are registered for execution, using the strategies presented in Subsection 2.1, and share scans if possible.

Figure 2 shows the system architecture. A client builds a query and sends it to the server, which executes all queries simultaneously and returns results gradually. Each client has an optional server process within the DB engine, which transfers the client commands to

Figure 2: FluxQuery Architecture

the engine. The server process is optional since if the client process is given access to the internal DBMS API, it becomes unnecessary to hold a dedicated server process (similar to *Shared Servers* [29]).

A single server process handles requests from multiple clients. Its responsibility is to transfer client commands to the appropriate join process for both query registration and query execution. A significant effect of this design is the ability to issue commands while the client is waiting for results of previously issued queries, resulting in a *many-to-many* relationship between queries and sessions (session is a connection between a client and a database server).

Each join process performs a cyclic join, enabling resource sharing between queries in order to provide the users their results as quickly as possible, minimizing the overall system overhead.

A major design decision was to send the results directly from the join process to the client, instead of through the server process. This architecture nullifies the need for result queues for inner server communication between the server process and the join process, which is done in the *Shared Servers* architecture.

We consider two key query operators in the interactive setting – selection and join. Although our approach is similar to that of Crescendo [17] and SharedDB [16], we cover joins in depth and provide techniques to decrease the response time by allowing queries to begin execution sooner.

3.2 System Design

Current commercial and traditional research DBMS systems are based on a threading model that assumes each process requires a latch, a short synchronization mechanism used to protect memory structures, on the data it uses. However, one of the basic concepts introduced in this paper is sharing latches and table scans among different queries. This design limitation leads us to write storage and query execution modules that allows the necessary sharing. While designing the storage structure, we used ideas for table and block design presented in [2, 12].

FluxQuery uses two main engines - Storage Engine (SE), and a Query Execution Engine (QEE), both were written in C++ (~6,000 lines) using pre-optimized libraries, e.g. Boost. The storage engine manages memory and disk space. When creating a new object, the SE assigns storage to the object. Each object is structured from a set of “database blocks” (or “blocks”) each of which is a sequential memory structure of a configurable block size. Each object can have a different block size, but all the blocks of a specific object are of the same size. The object size changes dynamically. Each database block contains records of variable sizes depending on the object definition. Blocks can be synchronized with the disk by us-

ing a buffer manager, which is not fully implemented in our system. The interface of the SE allows adding rows and deleting rows – the disk and memory management are obfuscated from the user.

The SE implements efficient latching and row locks, assuring that clients can query and modify records safely. In addition, the SE also provides “snapshot” capabilities for queries – allowing implementing multiple transaction visibility options. We used a block structure similar to the one employed by commercial systems in order to confidently compare their performances, and as shown in the evaluation, the performance of our naïve algorithms is similar to that of the commercial ones.

The QEE implementation uses streaming and pipelining. Each operator is implemented within a process that receives a set of streams and emits a single stream. Processes can be pipelined together to form the user’s query, e.g., 3 queries that join over two tables using the same join criteria will use the same join process. If for two queries additional subsetting is required using different conditions, an additional subsetting process will be created that will serve both. Architecturally, this scenario sums up to a join process that executes two queries - one resultset is sent to the user, while the other one is sent to an additional server process. The additional subsetting process executes both subsettings, although each is on a different column, and streams resultsets to the clients.

The system is agnostic to the number of incoming streams to each server process - each server process reads from multiple streams. Although we implemented a proof of concept for multi-table joins (join over three or more tables), we have not thoroughly researched its behavior and performance. We have not implemented a query optimizer. The execution engine accepts an already parsed execution plan for DMLs, data modifications, and queries.

3.3 Preliminaries

Multiple ways exist to share resources such as latches, connections, processes, etc. We favor two techniques – Shared Servers [29] and Shared Scans [16, 17].

As opposed to previous work, we focus on providing initial results within interactive response times. In the Shared Server approach, the CPU is assumed to be the congested resource and, hence, the access to it is controlled by the Shared Server processes by sharing the session among different processes; However, the data is not shared between different queries or sessions. It results in a decreased and controlled load on the system at the expense of increased response time. On the other hand, Shared Scans utilize the same table scan for multiple queries to provide results within a predictable interval. It introduces a significant delay between the query being introduced in the system and starting execution. Both approaches have drawbacks with respect to interactivity since the start of a query is delayed in both approaches.

An important challenge we face is improving interactivity without harming the overall query performance. We propose using shared latches within cyclic scans, with an interactive client-server communication protocol. Although the time a latch is held while a block is joined is linearly dependent on the number of queries, the duration of joining two blocks for one query or thousands of queries was less than 1ms in our setting (32KB block size, approximately 256 rows in each block). On the other hand, the time taken to obtain a latch varies from 1ms to 200ms (averages to 20ms) – due to contention, the time taken to obtain a latch increases with bigger system loads. Therefore, the main performance bottleneck is block latching, and not the join itself.

Shared latches, although developed for Cyclic Scans, can also be used to reduce the number of latches in a generic DBMS, helping to avoid contention problems while increasing the system stability and performance, issues that are hard to address.

3.4 Cyclic scan

The most commonly used joins are *Nested Loop Join (NL)* and *Hash Join (HJ)*, which we focus on in this paper and base our cyclic scan engines on. Using the cyclic scan approach [6, 16, 17, 21, 29, 52] reduces contentions and improves query response times, and can be easily implemented for join algorithms that scan the relations linearly. These approaches enable creating new join operators that fit the interactive scenarios better than current ones.

In our cyclic approach, a single thread in a cyclic manner continuously scans the *external table* (chosen by an optimizer). For each block of the external table, each block of the internal table is scanned or the relevant hashmap searched. When a query is issued, it does not start execution immediately. Instead, it registers itself in a pending queue of future queries to run. Query modifications such as cancellations and priority changes are performed in a similar fashion – the modification frequency is controlled by the client or the execution strategy used (Section 2). Depending on the granularity of the *safe points* (elaborated in Section 4.4, safe points can be briefly described as the logical points to modify the list of queries the join process operates on), we check if there are any pending queries that can start execution. If such queries exist, we start the execution of the new query from the current scan position, where its execution will also be stopped.

Algorithm 1 Naïve Algorithm

```

1: procedure NAIVEINTERACTIVEJOIN
2:   while true do
3:     for Each block of external table do
4:       for Each block of internal table do
5:         for Each external block row do
6:           for Each internal block row do
7:             Safe point - modify query execution list
8:           for Each query in current query set do
9:             Latch the blocks
10:          Perform join

```

The naïve algorithm, Algorithm 1, demonstrates the basic ideas of the cyclic join. A discussion regarding the placement of the safe point can be found in Subsection 4.4. An important optimization is obtaining the block latches for every block combination (between Lines 4 and 5), instead of for every row and query combination (Line 9). As noted before, the time taken to get a latch often greater than the time to scan the data, making it more practically efficient to obtain the latches for blocks instead of for rows.

The API (presented in Table 1), which is part of the protocol for our cyclic scan engine, and based on the interactivity requirements shown in Section 2, primarily consists of the following methods: RegisterQuery, ReadResult, ModifyQuery, and StopQuery. The commands RegisterQuery, ModifyQuery, and StopQuery are asynchronous and return control to the client almost immediately.

API	Description
RegisterQuery(q)	Registers the query q in the system for execution using the relevant join process.
ReadResult(q, s)	Reads the next result. If there are no results, the system waits for a result for s seconds.
ModifyQuery(q, m)	Modifies properties of q, such as priority and maximum number of tuples to send. m is a map of properties to modify.
StopQuery(q)	Removes query q from the execution list.

Table 1: FluxQuery API

4. ALGORITHMS

Algorithm	Description
Algorithm 2	Full execution thread for CNL
Algorithm 3	Full execution thread of CHJ
Algorithm 4	FluxJoin main concept
Algorithm 5	Fast FluxJoin main concept
Algorithm 6	Complex communication protocol for query registration for CHJ
Algorithm 7	The safe point of CHJ. Demonstrates the communication protocol

Table 2: Algorithm List

We introduce four different algorithms - *CNL* (Cyclic Nested Loop Join), *CHJ* (Cyclic Hash Join), *FJ* (FluxJoin), and a variation of it, *FFJ* (Fast FluxJoin). *FJ* and *FFJ* draw from both *CNL* and *CHJ*. They have a lower response time compared with *CHJ*, and a lower total execution time compared with *CNL*. Both are based on the observation that multiple queries can share the same hashtables/hashmaps built over the internal table, avoiding multiple scans of the internal table data for different queries. Instead of scanning over the internal table blocks to match rows to an external table block, we build a hashtable over the queried columns of the internal block to speed up lookup for matching rows, decreasing response time.

The query lists and the hashmaps are accessed only by the join execution thread. Hence, latches are only needed for data blocks and safe points (Subsection 4.4) on the thread executing the join.

Algorithm 2 Cyclic Nested Loop - Execute

```

1: procedure EXECUTE JOIN
2:   start :
3:   ▷ Waits for a new query to register from parallel threads
4:   PendingQueriesSemaphore.Wait()
5:   while true do
6:     for Each external table block exb do
7:       for Each internal table block inb do
8:         ▷ This is a safe point for modifying objects
9:         CNLSafePoint()
10:        ▷ Execution of Nested Loop:
11:        LatchAsReader(exb);
12:        LatchAsReader(inb);
13:        for Each row er in exb do
14:          for Each row ir in inb do
15:            for Each query q in Queries do
16:              if q.Join(er, ir) then
17:                emit q.JoinResults to client
18:                q.TotalResults += 1
19:          ReleaseLatch(inb);
20:        ReleaseLatch(exb);
```

In Algorithm 2, we show the *CNL* algorithm execution process. The algorithm demonstrates that in the absence of queries, the execution thread sleeps, while during query execution, the basic *NL* algorithm is executed for each query. The fact that each query begins and ends execution at the correct location is guaranteed by the safe point (Subsection 4.4).

In Algorithm 3, we show the execution thread for *CHJ*. In *CHJ*, the internal table blocks are not accessed directly as in *CNL* – the rows are accessed through the hashmaps. Thus, it does not iterate over the internal table. The hashmaps needed are assured to be available due to the safe point – the hashmaps are built within the

Algorithm 3 Cyclic Hash Join - Execute

```

1: procedure EXECUTE JOIN
2:   start :
3:   ▷ Waits for a new query to register from parallel threads
4:   PendingQueriesSemaphore.Wait()
5:   while true do
6:     for Each external table block exb do
7:       ▷ This is a safe point for modifying objects
8:       CHJSafePoint()
9:       ▷ Execution of Hash Join:
10:      LatchAsReader(exb);
11:      for Each row er in exb do
12:        for Each query q in Queries do
13:          map = CurrentHashMaps.get(q.HashmapCol)
14:          if q.Join(er, map) then
15:            emit q.JoinResults to client
16:            q.TotalResults += q.JoinResults.size
17:      ReleaseLatch(exb);
```

safe point since multiple queries might use the same hashmaps (see later discussion).

4.1 FJ - The FluxJoin

FluxJoin is designed to be used in a cyclic fashion. *FJ* is based on the observation that the Nested Loop Join is extremely responsive, but slow, while the Hash Join is fast, but slower to respond. Furthermore, data blocks are independent of each other but are ordered for the purpose of a table scan. We developed a combination of the two algorithms, Nested Loop Join and Hash Join, that allows the usage of hashmaps for accelerating the Nested Loop Join – *FJ*. This was done by changing the granularity of the hashmap from being on the full table to being on a block.

We reuse the block-hashmaps only for each external-internal block combination, and do not cache them in order to limit the memory dependency. Therefore, it is able to handle the cases where *CHJ* cannot fit all the hashmaps in memory. We have not considered using multiple blocks for building the block hashmaps – an optimization option that might be beneficial. Determining the optimal number of blocks for such a scenario can be done using a variety of methods such as auto-tuning [25] or workload profiling, and is ideal future work.

In *FJ*, as seen in Algorithm 4, we execute a modified *CNL*, in which in the innermost iteration, we do not iterate over each row within the block. Instead, we build hashmaps on the current inner block for each column that is queried, and search it using the external block rows. We reuse the same hashmaps for all the queries on the same column, as done in *CHJ*. Maps are held only for the duration of the internal iteration.

Figure 3 illustrates the steps taken by *FJ*. We assume that 3 queries are running and join using 2 columns of the inner table. While the cyclic scan is executing (Step 1), we build a set of hashmaps on all the queried columns for the current internal table block (Step 2). After the hashmaps are built, all the queries are executed (Step 3). Next, the blocks are switched and the process repeats (Step 4).

For *q* queries that use *c* distinct columns, if the outer relation has *o* blocks with *m* rows each and the inner relation has *i* blocks with *n* rows each, the complexity for *CNL* would be $O(o \times m \times i \times n \times q)$, while for *FJ* the complexity would be $O(o \times i \times (n \times c + m \times q))$. Thus, we can see that in the case where numerous queries use the same columns, the performance gain of *FJ* is greater than that of

Algorithm 4 The FluxJoin

```

1: procedure EXECUTEFJ
2:   start :
3:   ▷ Waits for a new query to register from parallel threads
4:   PendingQueriesSemaphore.Wait()
5:   while true do
6:     for Each block of external table eb do
7:       for Each block of internal table ib do
8:         ▷ This is a safe point for modifying objects
9:         CNLSafePoint()
10:        ▷ Execution of FluxJoin:
11:        LatchAsReader(exb);
12:        LatchAsReader(inb);
13:        maps = BuildMapsForBlock(ib, Queries)
14:        for Each row in eb do
15:          for Each query in Queries do
16:            Join row with column map
17:            Emit Results
18:          ReleaseLatch(inb);
19:          ReleaseLatch(exb);
20:          Free(maps)

```

Figure 3: FluxJoin Walkthrough

CNL. In the worst case where $c = q$, FJ's performance is similar to that of CNL because m and n are small and limited.

In contrast with Hash Join and Nested Loop Join, assuming the number of rows in each block of both the tables are similar (most databases use the same block size for all tables), it is not important if the internal table, the one on which we build the hashmaps, is the smaller table or the bigger table. In both cases, the complexity and response time are not affected.

FJ consumes a constant and low amount of memory compared with Hash Join. The size of all hashmaps for a data block cannot be greater than the size of nine data blocks and would rarely exceed the size of two data blocks. Each distinct value that appears in a data block will occupy its size in the hashmap, and in addition the hashmap holds a set of pointers to the data within the data block. Assuming that in the worst case there are no repetitions of any values, the values would require at most the size of one data block. Each value will also have a pointer. Supposing each value is at least one byte and a pointer is at most eight bytes, the hashmap size will not exceed the size of nine data blocks. This worst case analysis

is extreme, since most values held are bigger than a byte. Further, due to repeating values in databases, the size of the FJ hashmaps would often be equal to the size of one to two data blocks.

FJ's memory overhead is negligible and fits well within the different levels of system caches. The performance of FJ is similar to that of CNL when a single query is executed, yet, when multiple joins execute using the same column, the performance improvement is linear with regards to the number of executing queries thanks to hashmap sharing. FJ cannot perform as well as HJ due to repeated creation of hashmaps for the inner table blocks. However, our experiments show that when thousands of joins are executed over the same set of columns, FJ's performance is closer to that of HJ than to that of NL because of the hashmaps reuse. Considering that FJ emits results nearly as fast as NL, it is ideal for interactivity.

4.2 FFJ - The Fast FluxJoin

FJ does not reuse hashmaps across different external blocks, and thus can be optimized further. In FFJ we assume all hashmaps can fit in memory, and build full hashmaps for each queried column while FJ executes and emits results – preserving its CNL-like low response time for initial results and achieving CHJ-like low total execution time. After the full hashmap is built on each column, which happens at the second external table block after adding a query, the join process can use it instead of scanning the internal table. This limits the number of internal table scans to the number of columns, in the worst case.

Algorithm 5 The Fast FluxJoin

```

1: procedure EXECUTEFFJ
2:   for Each block of external table eb do
3:     Execute a Safe Point
4:     for Each query with prepared hashmap do
5:       map = FullMaps[column]
6:       Emit results of eb joined with map
7:     if maps need to be built then
8:       for Each block of internal table ib do
9:         Execute a Safe Point
10:        for Each column that needs a hashmap do
11:          map = BuildMapForBlock(ib, column)
12:          FullMaps[column].merge(map)
13:        for Each query on column do
14:          Emit results of eb joined with map
15:        Free(map)

```

In Algorithm 5, for brevity, we show the high-level design of the Fast FluxJoin (FFJ) algorithm, and have not shown sleeping or latching mechanisms. In FFJ, when a new query begins execution, the hashmap needed for it on the internal table either exists or does not. If the needed hashmap is fully built, we execute Cyclic Hash Join. If not, we execute FluxJoin and while doing so, we build a full hashmap that will be used for later hashjoins. The safe point has to be modified in this situation for considering the case where a hashtable is partially built.

In the case of partially built hashmap, within the safe point we execute Hash Join by using the partially built map, and then continue executing FluxJoin using the rest of the internal table blocks – together the entire internal table is scanned for the specific external block. Starting with the next external table block, the fully built hashmap will be used. This behavior is similar to that of the Ripple Hash Join [22], jumping back to already scanned data for deferred joining with the already seen data, yet both are substantially different as follows – 1. FFJ scans both tables linearly, and the *jump* to

previous already scanned data is limited only for the internal relation, using a hashmap. 2. The usage of the partially built hashmap will happen at most once for a query, and only while the hashmap is not yet built.

The benefits of this approach are substantial, and as shown later, the performance is similar to CHJ, while providing interactive response time. In FJ, for each block of the internal table, we have to build the hashmaps repeatedly. In comparison, in FFJ, we scan each block of the internal table once for each queried column.

The complexity of FFJ for the average case is (as given before, there exist q queries, c distinct columns, outer relation has o blocks with m rows each, and inner relation has i blocks with n rows each) $O(c \times i \times n + i \times m \times q + o \times m \times q)$. The first term indicates the cost of building the hashmaps on the internal tables. The second term shows the complexity of joining the first external table block with every block in the internal table (note the term o that appeared in the complexity for FJ is missing here). The last term indicates the cost of joining the rest of the external table blocks with the fully built hashmaps. Thus, we can note that by having enough memory to hold the entire hashmaps, we are able to achieve execution times similar to those of CHJ while obtaining CNL-like response time.

4.3 Comparison between Join Algorithms

Each join algorithm mentioned above will be useful for a different scenario. Here we summarize the preferred use case for each:

- CHJ is preferable when all results are necessary, and there is no constraint on response time. CHJ builds a full hashmap on one of the tables, if it wasn't built and cached earlier, and until this step is completed no results will be emitted. The caching of the hashmaps results in excess memory consumption, and if memory is limited it is not feasible.
- CNL produces results in an extremely low response time. Yet, FJ or FFJ are nearly always preferable over CNL since both always perform the same, or better, than CNL.
- FJ is preferable when memory is limited and interactive response time is needed.
- FFJ is preferable when memory is not limited, but an interactive response time is needed. This algorithm builds hashmaps and caches these similarly to CHJ.

4.4 Safe points

We define a safe point to be the place within the cyclic join where we can freeze the join execution process and modify the list of executing queries safely – add a query, remove a query, or change query properties.

The optimal place for the safe point is just before latching the data blocks since it shortens the latch times – the safe point is entered often enough for query modifications to not delay the query execution start substantially and barely affects already running queries (a delay of less than 5ms for query to start, and less than 3ms for currently executing queries).

Since query registration requires the join process itself to freeze in order to prevent data races, we execute the safe point on the same thread as the join itself – a design that allows us to use safe *change lists* and unsafe, fast, join process internal lists. There are two obvious places to freeze the join execution in order to register queries – the switch of a block in the external table, which we define as Outer Registration Window (ORW), and the switch of a block in the internal table, which we define as Inner Registration Window (IRW).

In IRW, for every internal-external block combination, before executing the join for the already registered queries, we freeze the join and check whether any new queries exist that can start execution, or if any query exists that has completed a full cycle.

In ORW, instead of freezing the join process in order to check for new queries and scan completions for every outer-inner block combination, we do so only for outer blocks. Therefore, we scan the entire internal table before registering the pending queries.

Thus, we can see that in IRW, each query starts running virtually immediately after its registration – but the external table scan is paused more often. However, in ORW it takes a full scan for the internal table, in the worst case, for a query to be scheduled to run. This results in fewer stalls for the already executing queries. The choice between using IRW and ORW is therefore based on the tradeoff between providing interactivity and reducing the overall query execution time.

It is trivial that IRW is preferable for CNL (it reduces the delay between query registration and execution). Yet, ORW is preferable for CHJ because of the extremely short lookup time of records in the hashmap and the limited number of rows in a database block.

Algorithm 6 Cyclic Hash Join - Add Query

```

1: procedure ADD_QUERY(query)
2:   col = Column that should be hashed by the query
3:   CAS_Lock(ColumnsHavingHashes)
4:   if !ColumnsHavingHashes.Contains(col) then
5:     if !CurrentlyBeingBuilt.Contains(col) then
6:       SafeAddToList(CurrentlyBeingBuilt, col)
7:       CAS_Unlock(ColumnsHavingHashes)
8:       map = CreateHashmap(col)
9:       SafeAddQueue(PendingMaps, map)
10:    else
11:      CAS_Unlock(ColumnsHavingHashes)
12:      sem = new Semaphore
13:      SafeAddQueue(PendQuerySem[col], sem)
14:      sem.wait()
15:    else
16:      CAS_Unlock(ColumnsHavingHashes)
17:      SafeAddQueue(QueriesToAdd, query)
18:      CAS_Lock(QueriesToRemove)
19:      if CAS_Read(ExecutingQueriesCounter) == 0 then
20:        PendingQuerySemaphore.Post()
21:      CAS_Unlock(QueriesToRemove)

```

In Algorithms 6 (CHJ Add Query) and 7 (CHJ Safe Point), we demonstrate the communication protocol between a client request to add a query and the server process. In the Add Query algorithm, there are three scenarios that need to be considered – the hashmap needed for the query exists, it does not exist and needs to be built, or it is currently in the process of being built. If the hashmap exists, we simply use it. If the hashmap does not exist, the first client thread that needs the hashmap builds it. After the hashmap is built, it is added to the pending hashmap list (the modification of the actual hashmap list is done by the join thread itself within the safe point as demonstrated in Algorithm 7, to avoid the need for latching). If the query to be registered needs a hashmap that is currently being built, the thread that needs it goes to sleep after registering itself as pending for that specific hashmap. This allows the join executor thread, within the safe point, to wake up the sleeping thread when the hashmap is ready. For simplicity, we have presented a synchronous algorithm for adding queries, although, as mentioned before, it can be easily modified to be asynchronous.

In addition to the complex hashmap management, this protocol allows the process which executes the join to *sleep* while waiting for requests. A part of the Add Query algorithm, and its synchronization with the safe point itself, allows waking up the executor thread if no query was executing, saving system resources.

Algorithm 7 Cyclic Hash Join - Safe Point

```

1: procedure CHJSAFEPOINT
2:   for Each query in Queries do
3:     ▷ Remove queries that are done
4:     if query.ResultsCount ≥ MaxResults OR
5:       query.ShouldStop(exb) then
6:       SafeAddQueue(QueriesToRemove, query)
7:   if !QueriesToAdd.empty() then
8:     ▷ Queries are waiting to be added to the run queue
9:     Lock(QueriesToAdd)
10:    while map = SafeDequeue(PendingMaps) do
11:      col = Column the HashMap hash is on
12:      CurrentHashMaps.add(col, map)
13:      for Each semaphore in PendQuerySem[col] do
14:        semaphore.Post()
15:    while !QueriesToAdd.empty() do
16:      query = SafeDequeue(QueriesToAdd)
17:      AddToList(Queries, query)
18:      CAS_Add(ExecutingQueriesCounter, 1);
19:      Unlock(QueriesToAdd)
20:   if !QueriesToRemove.empty() then
21:     ▷ Queries are queued to finish
22:     Lock(QueriesToRemove)
23:     while !QueriesToRemove.empty() do
24:       query = SafeDequeue(QueriesToRemove)
25:       RemoveFromList(Queries, query)
26:       CAS_Sub(ExecutingQueriesCounter, 1);
27:       Unlock(QueriesToRemove)
28:   if ExecutingQueriesCounter == 0 then
29:     go to start

```

Line 5 of Algorithm 7 decides if a query should stop execution based only on *exb*, the external block. This signifies that this specific CHJ algorithm implements ORW. For CNL implementing IRW this method would require the internal block, *inb*, as well.

The safe point protocol for CNL is much simpler than the one demonstrated here. Although it implements IRW, the “Add Query” only includes adding a pending query while the safe point itself only manages the execution list from within the safe point itself – there is no need to manage hashmaps there.

Currently, we keep the hashmaps in memory until the join process terminates – the safe point does not clear hashmaps with completion of queries. It is possible to release a hashmap from memory or save it to disk after all the queries that use it have completed, which we plan to do in the future by using different eviction policies such as LRU, ARC, etc. FluxJoin, presented in subsection 4.1, avoids memory exhaustion and is ideal in settings requiring both - limited memory and interactivity.

An optimistic method to test for pending queries, first without latches, and then with latches on the internal data structures and queues, improves performance and makes the check for the pending modifications nearly free. The correctness of this approach lies in memory fences imposed by the latches on the data blocks. The penalty for using the optimistic approach is that in the worst case, a registered query would have to wait for two block cycles, instead of one. Paying this price is reasonable since block switches in a scan are more frequent than query modifications, and the stall duration is short and lasts for at most a few milliseconds.

Our preliminary experimental results aligned with our intuition and both directed us to these conclusions:

- For CNL, the safe point should implement IRW. The performance penalty is relatively small compared to the delay in start of query execution, which ORW imposes.
- For CHJ, the safe point should be at ORW.
- For FJ, the safe point should be at IRW, similarly to CNL.
- For FFJ, the safe point should depend on the current state - while building the hashmaps, IRW should be implemented, after the hashmaps are built, ORW is preferred.
- An optimistic method for modification of the execution list needs to be implemented for allowing safe point to be executed often at a low cost.

5. EVALUATION

We evaluate our system’s scalability and response time, i.e. interactivity. We measure our system performance over increasing datasets (table content of between 10KB and 100GB) and increasing number of queries (between 1 to 2048), while determining whether each experiment met the interactivity constraints. We demonstrate the scalability of our system – all our algorithms successfully process thousands of simultaneous queries, while FJ and FFJ provide results in interactive response time.

5.1 Experimental Setup

We performed our experiments on the following configurations: MACHINE_{244GB} : RedHat (kernel 2.6), Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz x 4 (32 cores), 20 MB L2 cache with 244 GB of RAM (EC2 `r3.8xlarge`), MACHINE_{4GB} : CentOS (kernel 2.6), Intel(R) Xeon(R) CPU X3210 @ 2.13GHz x 1 (4 cores), 8 MB L2 cache with 4 GB of RAM.

Datasets: Since a benchmark for interactivity with large datasets does not exist, we synthesized data in such a way as to control join selectivity. In Table 3, we provide the selectivity for some data sets, all based on TPC-DS [40] – between 3% to 10%. The *Selectivity* describes the percent of data selected from each joined table, while the *Cartesian selectivity* denotes the selectivity of the resultset compared to a cartesian multiplication of both tables. Unless reported otherwise, the number of columns in each table is set to 16. Between table sizes of 10 KB to 100 GB, our experiments cover the entire spectrum of expected requirements from in-memory DBMS’s.

Selectivity	100MBx10MB	1GBx1GB	100GBx100GB
Table	4.11%	4.11%	2.05%
Cartesian	0.1%	0.01%	0.005%

Table 3: Data Sets Sizes Selectivity Setting

Workloads: The workloads determine how many queries will be executed in parallel and when each begins and ends execution. The workload is orthogonal to the size of resultset – there is no limitation on the resultset size. We constructed 2 workloads based on different burstiness behaviors (Section 2.1) of joins involving 2 tables– WORKLOAD_{FULL} (default) and WORKLOAD_{TOPK}. We also experimented with a real workload, provided by the authors of [35], WORKLOAD_{REAL}. In this workload, users gesturally interact with their system to build simple queries.

The probabilities that the *TOPK* algorithm uses are based on real users interaction with a demo system, described later in Subsection 5.3.5. The *PDF* changes over time based on the user interaction with the system.

Baseline DBMS: We compared our system with System X, a commercial DBMS system; System Y, a commercial vector based DBMS system based on [5]; and PostgreSQL 9.3. We focused on these systems since they are intended to be used in data warehouses and represent the majority of research and commercial systems.

Although System Y outperformed System X for queries involving a single relation, System X outperformed System Y in every experiment involving joins. Hence, we have not reported the performance of System Y.

We were unable to obtain neither implementation, source code, nor executables of research DBMS engines more closely related to ours despite multiple attempts to contact authors of multiple papers. As a solution, we discuss the design principles for all related work in the absence of these specific implementations.

5.2 Metrics

Each experiment was repeated 3 times, after a warm-up run, and we report the average of the 3 runs in logarithmic scale. We report the standard deviation (*stdev*) where applicable. In all Hash Join experiments, the hashes are neither cached nor re-used across runs.

Interactivity Constraint Criteria: For each experiment, we tested whether the results were returned within an interactive response time limit, set to 0.5 Seconds. This value was chosen as the *interactive time* criteria since it has been shown to be the average human response time [41] and the detrimental effects of latencies over 500ms have been well demonstrated [31].

We set the minimum number of results to be returned to be 500 in order to consider a system as interactive so as to make sure that a reasonable number of tuples are available for the user to skim – unlike most related work, which sets it at a single tuple. Although the interactivity criteria requires us to deliver limited amount of results in limited time, we do not stop query execution after these were produced unless the clients requests so.

Time for Completion: For each experiment, we report the execution time for all queries in seconds, which represents the time between the client issuing its first query and receiving all the results for all issued queries. This time is unrelated to the interactivity criteria, which is a binary criterion.

5.3 Experiments

5.3.1 Scale Of Simultaneous Queries

Traditional systems are limited in the amount of parallel computations they can provide since different clients use separate threads. These systems utilize the idle time spent waiting for resources needed by a user to process other requests. If idle time is unavailable, they get overloaded. In our experiments, PostgreSQL and System X could not process more than 128 and 512 parallel requests respectively. In contrast, our system was able to reach the maximum available threads allowed by the operating system.

In Figures 4 and 5, we compared the query execution times for different relation sizes with increasing number of concurrent queries. We have not reported the performance of NL or CNL here since they are much slower than CHJ. We have presented the execution times for FJ and FFJ only for $\text{Data}_{100MB \times 10MB}$, since for $\text{Data}_{1GB \times 1GB}$ adding FJ distorts the graph due to a scale change – the experiments provided later compare the cyclic algorithms specifically, while Figure 5 compares hash join only.

The *stdev* of the execution times across the different runs differs based on the data set. For $\text{Data}_{1GB \times 1GB}$, the *stdevs* of CHJ, System X, and PostgreSQL are 0.05s, 223.13s, and 326.95s respectively. For $\text{Data}_{100MB \times 10MB}$, the *stdevs* of CHJ, FFJ, FJ, and System X are 1.10s, 11.61, 16.03s, and 281.01s respectively. The *stdev* increases with increasing number of threads for all experi-

Figure 4: Time To Completion, $\text{Data}_{100MB \times 10MB}$, Machine_{4GB} . Here, CHJ has the least execution time, while FJ scales the best (smallest slope).

Figure 5: Time To Completion, $\text{Data}_{1GB \times 1GB}$, Machine_{4GB} . Here, CHJ is the only algorithm that scales, while having the least execution time.

ments due to increase in both the query execution times as well as the overall system load.

We conclude:

- CHJ offers the best scalability.
- The performance of FJ is comparable to some of the hash join implementations.
- The performance of FFJ is closer to CHJ than to any of the other cyclic join algorithms.
- The cyclic algorithms have better performance predictability (lower *stdev*).

5.3.2 Impact of Data Size

We report whether each algorithm can be considered as interactive in Tables 4, 5, and 6 based on the interactivity criteria (Section 5.2) for data sizes ranging from 10 MB to 100 GB. '✓' and '✗' specify whether the experiment held the criteria or not respectively. '.' denotes that the underlying DBMS failed execution.

HJ does not provide interactive response times in most of the cases, while FJ, FFJ, and NL usually do (for our engine, NL, CNL, and FJ always provides interactive response times, while some of the systems we compared to did not). The cyclic versions of all

Threads	16	32	64	128	256	512	1K	2K
HJ	✓	✗	✗	✗	✗	✗	✗	✗
System X HJ	✗	✗	✗	✗	✗	✗	·	·
PostgreSQL HJ	✗	✗	✗	✗	·	·	·	·
CHJ	✓	✓	✗	✗	✗	✗	✗	✗
System X NL	✓	✓	✓	✓	✓	✓	·	·
PostgreSQL NL	✗	✗	✗	✗	·	·	·	·
NL	✓	✓	✓	✓	✓	✓	✓	✓
CNL	✓	✓	✓	✓	✓	✓	✓	✓
FJ	✓	✓	✓	✓	✓	✓	✓	✓
FFJ	✓	✓	✓	✓	✓	✓	✓	✓

Table 4: Interactivity Over Data_{100MB×10MB}, Machine_{4GB} for simultaneous queries (threads). Here, NL, CNL, FJ and FFJ are the only interactive algorithms.

Threads	16	32	64	128	256	512	1K	2K
HJ	✗	✗	✗	✗	✗	·	·	·
System X HJ	✗	✗	✗	✗	✗	✗	·	·
CHJ	✗	✗	✗	✗	✗	✗	✗	✗
System X NL	✓	✓	✗	✗	✗	✗	·	·
NL	✓	✓	✓	✓	✓	✓	✓	✓
FJ	✓	✓	✓	✓	✓	✓	✓	✓
FFJ	✓	✓	✓	✓	✓	✓	✓	✓

Table 5: Interactivity Over Data_{1GB×1GB}, Machine_{4GB} for simultaneous queries (threads). Here, NL, FJ, and FFJ are the only interactive algorithms.

the algorithms were able to handle a heavy load (up to 16K simultaneous queries), while their simplistic counterparts, the HJ and NL, failed much earlier. Interestingly, when HJ failed, the reuse of cached hashmaps allowed CHJ to complete.

All the hash joins (including FFJ) require additional memory during execution. This has a detrimental effect on the experiments performed using Data_{100GB×100GB} even on machines such as Machine_{244GB} – the entire memory is consumed and the hash join execution fails. In comparison, CNL does not consume additional memory, while FJ requires limited additional amount of memory. As seen in our experiments, FJ and FFJ always provide interactive response time.

We conclude:

- The cyclic versions are preferable over the non-cyclic ones for both performance as well as interactivity.
- FJ, FFJ, and CNL are always interactive.

5.3.3 Impact of Workload Strategy

We set a limit of 10 seconds for an interaction with the system to complete, i.e. after 10 seconds, only the user-intended query remains. On Data_{100GB×100GB}, both CNL and FJ provide response within interactive times, although the user query took more than 10 minutes to complete. CHJ and FFJ could not finish execution since they required more memory than was available.

	CHJ	FFJ	FJ	CNL
Workload _{TOPK}	1.97S	16.94	29.97S	404.25
Workload _{FULL}	2.07S	16.85	32.79S	6168.36

Table 7: Impact of workload on execution time, Data_{100MB×10MB}, K=512. Here, Workload_{TOPK} has better performance than Workload_{FULL}.

Threads	64	128	256	512	1K	4K	8K	16K
HJ	·	·	·	·	·	·	·	·
System X HJ	✗	✗	✗	✗	✗	·	·	·
CHJ	·	·	·	·	·	·	·	·
System X NL	✓	✓	✗	✗	✗	·	·	·
NL	✓	✓	✓	✓	✓	·	·	·
CNL	✓	✓	✓	✓	✓	✓	✓	✓
FJ	✓	✓	✓	✓	✓	✓	✓	✓
FFJ	✓	✓	✓	✓	✓	✓	✓	✓

Table 6: Interactivity Over Data_{100GB×100GB}, Machine_{244GB} for simultaneous queries (threads). Here, CNL, FJ, and FFJ are the only algorithms that scale for large relations, while being interactive.

Figure 6: Join Types Comparison, Data_{10MB×10MB}, Machine_{4GB}. Here, FJ and FFJ represent a good trade-off between CHJ (non-interactive) and CNL (interactive).

In Table 7, we show the impact of the execution strategy on the execution time. All queries returned results within our interactive time limit. K was chosen to enable high query load as well as high query flux (queries are added and removed from the top-k list frequently, approximately every 10 – 100ms).

Using the TOP-K technique reduces the system overhead for CHJ and FJ, while holding the interactivity criteria, saving approximately 10% of the total execution time. Although a larger number of queries are sent to the DBMS in the top-K approach (since some queries need to be stopped and restarted), the performance of these algorithms improves. FFJ was expected to execute slower for the Top-K workload, because query removals and additions make it fluctuate between CHJ and FJ – an expensive operation. However, no substantial performance change was observed.

We conclude:

- Top-K workloads improve performance for CHJ, FJ, and CNL.
- FFJ is not susceptible to the workload in use, whereas CNL is greatly affected.

5.3.4 Impact of Fast FluxJoin and FluxJoin

We compare the behavior of FJ and FFJ with CNL and CHJ using Data_{10MB×10MB} to allow all the queries to finish execution.

In Figure 6, we show FJ is exponentially faster than CNL, yet slower than CHJ – making it a good intermediate point between the two when interactivity is needed. FFJ performs similarly to CHJ, and has the same performance pattern. Due to long execution

Figure 7: Join Types Comparison, Data_{1024MB} × 100MB on Machine_{244GB}. Here, FFJ is ideal for interactivity.

Figure 8: Average Query Time, Data_{10MB} × 10MB, Machine_{4GB}. Here, the average query time of FJ and FFJ are closer to the HJ family than the NL one, while FFJ is within the same scale as HJ and close to CHJ.

time, we ran CNL for select few configurations. In Figure 7, we show that FFJ performs nearly as well as CHJ, and better than FJ. Yet, FFJ provides interactive response time like FJ does, while CHJ does not. Figure 8 shows that cyclic joins have lower average query execution time than traditional joins due to resource sharing. The results using bigger data sets followed a similar pattern.

We conclude:

- All cyclic methods improve the execution time with increasing load for interactive workloads.
- FJ and FFJ are preferable over CNL since its performance is at worst similar to NL, when hashmaps cannot be reused.
- When memory is not limited, FFJ is the ideal algorithm - based on response time and performance.
- CHJ always has the least execution time.

5.3.5 Real workload

We experimented using *WORKLOAD_{REAL}* with table sizes varying from 10 KB to 100 GB and number of columns from 3 to 16. Since the systems we compared to rarely returned any results during the interaction, we only report our system's performance.

CHJ did not provide results within interactive times in any of the experiments with table sizes bigger than 100 MB, although it was always the fastest overall. FFJ performed similarly to CHJ, but held the interactive response time criteria. CNL, FJ, and FFJ were interactive for all queries in all the experiments, including the one involving two 16 column tables of size 100GB each.

The response time for the first 500 results across all the experiments for CNL, FJ, and FFJ was lesser than the interactive response time limit (average being 0.26s for CNL and FFJ, and 0.27 for FJ). For this workload as well, CNL and FJ have similar performance while running a single query and FJ performs better when it can reuse hashmaps, as the case of multiple queries (through the interaction duration). FFJ performs similarly to CHJ, but little slower.

Join Type	CHJ	FFJ	CNL	FJ
Increase Times	491.7%	385.9%	81.9%	81.8%

Table 8: Increase Percent Average. Additional queries add a heavy load to CHJ and FFJ, compared with CNL and FJ.

In Table 8, we present the average of the time difference across all data sizes, while varying the number of columns in the internal table from 3 to 16. Increasing the number of columns in a table results in decreased number of rows since the table size remains constant. We can see that the performance of CHJ worsened when the number of columns increases, while the performance of both CNL and FJ improved. FFJ performance is similar to that of CHJ, which deteriorated since more hashmaps were needed to be built, whereas FJ was able to reuse the hashmaps and CNL was better since it had to process a fewer number of rows. We conclude:

- The overhead of CHJ is high. When the number of columns increases, CHJ takes longer to execute while CNL and FJ perform better.
- FJ and CNL behave similarly when a single query is executed. As shown before, the advantages of FJ are more noticeable when multiple queries use the same hashmaps.
- FFJ has similar overhead to CHJ, yet provides interactive response time for the initial resultset.
- After user interaction ends, the performance of each cyclic algorithm is similar to its underlying algorithm, showing that the overhead of the cyclic implementation itself is low.

6. RELATED WORK

Interactive queries have been explained in [8, 24] and in [45, 46], the authors delve into details regarding the need for this type of queries in more depth than here. An interactive engine challenges the current state of the art systems with additional requirements such as limited response time and in-flux queries [14, 23, 50]. The current state of the art developments can be divided into two research areas – parallel scans and shared scans.

Parallel scans [3, 15, 18, 49, 51] aim to improve the resource utilization. The main idea is to reduce the system response time by using available resources that are not utilized. Although we have not explored parallelization, a parallel cyclic scan, or a parallel join on latched blocks, it is an interesting avenue for future work.

Shared Servers [29] allow different queries to execute within the same host server process. It also enables transactions to be executed on different server processes while preserving ACID requirements. It is implemented using queues that register queries for execution. Each query sequentially uses the Shared Server process. In Shared Servers, one query executes at a time within each Shared Server process. When the current query completes, the next query is dequeued and begins execution. In contrast with Shared Servers,

FluxQuery executes all the queries simultaneously, sharing the table scans, and not only the hosting process. The similarity between our approach and Shared Servers is mostly related to the queue for communication between the client and the execution process.

Shared scan, introduced explicitly in [16, 17] and based on [6, 21, 29, 52], introduces the ability to share a table scan for processing multiple queries. We adopted the shared scan approach since it is well suited for providing interactivity. The shared scan, as presented before, aims to provide system predictability in OLTP environments, whereas we focus on data warehouse environments. It originally targets a response time of 2s, which is greater than interactive thresholds of at most 0.5s. We add to the existing work a thorough discussion regarding joins including Cyclic Hash Join and FluxJoin, hitherto missing. We demonstrate how sharing latches in addition to scans can further speed-up query execution. In addition, we utilize the client for some of the server process operations (for example, building hashmaps). Furthermore, we start executing queries at the granularity of data blocks as opposed to the entire relation.

Multi-Column Joins are an interesting avenue to pursue. The techniques shown in [37] allow implementing one join process that enables two or more column join. The same approach, as presented there can be extended to more than three columns, therefore we believe the expansion of our CHJ/CNL algorithms to join more than two columns is feasible, and can be targeted in future work.

We implemented 4 different cyclic join algorithms: CNL, CHJ, FJ, and FFJ. Other join types such as sort-merge joins, etc. and aggregation types exist, which might also benefit from our approach. Some, for example ripple joins [22], cannot be implemented in a cyclic manner since data is not scanned linearly in both tables.

The database engines, objects, and operators change over time as shown in [30, 37]. Our objects are built in a manner that allows tunneling an output resultset stream into another operator (pipelining), allowing our system to use techniques shown in [53]. Cyclic scans need to be researched in more depth with regards to pipelining. We find query vectorization and pipelining techniques, presented in MonetDB [5], useful towards that goal.

In Eddies [1], *moments of symmetry* is introduced which has resemblance with our notion of *Safe Points*. Both are similar in the action they perform - enforcing a halt of the main algorithm for communication (in Safe Points, query list modifications, in “Eddies” for join data intended for processing). However, the intention and implementation are vastly different. Safe Points are meant for internal lists modifications, and are used for assuring consistency of memory structures, allowing multiple simultaneous queries execution while lowering performance hit. *Moments of Symmetry* are meant for synchronization of data between nodes processing a singular distributed join assuring result correctness.

Shared Operators have been introduced before in the context of continuous queries over streams [33]. Assuming the subsetting criteria is identical, Shared operators allow executing specific filters simultaneously for multiple queries. In our work, not only are the join criteria different, but the columns the joins are on differ as well - preventing us from using Shared Operators. Our system is structured for using different subsettings and join criteria while sharing the table scans.

7. CONCLUSION

In this paper, we formalized the concept of an in-flux interactive query. We discussed the different aspects and expectations a user might have from an interactive engine such as response time limits, system load, etc.

We introduced new techniques for joining tables while also sharing the data and the latches. We developed protocols for adding queries to the shared join process, while other queries are executed, at a low cost.

We proposed a hybrid join strategy called the FluxJoin which reduces the complexity of the Nested Loop Join, while using limited memory and maintaining the short response time necessary for interactive queries. Fast FluxJoin, a variant of FluxJoin, which results in lower execution time at the cost of additional memory, is shown as well.

We showed our system provides significant performance gain. The systems we compared with crashed when pressured with a heavy load. On the other hand, our engine’s average execution time scaled sub-linearly with increasing number of queries – more queries results in a larger degree of resource sharing.

The methods introduced in this paper can be implemented as optional elements in a full-featured database system, enabling clients to choose sharing the scan resources in exchange for interactivity, thus, reducing the system load.

8. FUTURE WORK

The query intent model has been introduced and not presented in great depth since the cyclic scan based system is the focus of this paper. It helps us motivate our problem setting and helps explain some of the FluxQuery design choices. The model deserves work with sole focus on it.

In the current implementation, we execute the join within the same process for all of the registered queries. Some of this work can be parallelized, such as emitting the results to the client, or performing the join itself for different queries.

When numerous results are produced rapidly, a *slow reader* should not exhaust the server resources, which in our case is the memory. Multiple solutions exist such as freezing sending of results to a slow reader until its buffer is cleared, improving the result stream object, or sending the most *interesting* results to the client, which decreases the required buffer size.

A major performance bottleneck was the transfer of the result set to the client. Work related to processing data streams [13, 47] can be used for handling this issue. It can be mitigated using sampling as well [9, 11, 38, 39, 44, 54]. Fault tolerance and scale out streams [7] are also relevant to our work.

Multi-table and multi-column joins are an interesting avenue of future work. Currently, multi-table joins can be implemented by pipelining multiple join processes while multi-column join can be implemented by one join process and additional pipelined subsettings. An interesting goal of implementing multi-table and multi-column joins within one process will be targeted in the future.

There is a clear need to research optimization of cyclic engines. In this paper, we have assumed that an optimizer, which builds execution plans for such engines exists, and we would like to work on it in the future – we currently manually built the execution plans.

There are multiple aspects of sampling that can be used in the context of interactive query execution. It is clear that a row sampled while keeping a particular join combination in consideration is usually not useful for another join combination and, thus, sampling rows with multiple join criteria in mind is an interesting avenue for future work. Finding samples that would be relevant for all possible join and subsetting criteria is both challenging and interesting.

Acknowledgements

We acknowledge the generous support of Amazon’s AWS Cloud Credits for Research Program and the U.S. National Science Foundation under awards IIS-1422977 and CAREER IIS-1453582.

9. REFERENCES

- [1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*. ACM, 2000.
- [2] E. Bakke. Schema-independent DB UI. *CIDR*, 2011.
- [3] D. Bayliss et al. Query scheduling in a parallel-processing database system, 2007. US Patent 7,185,003.
- [4] J. Bernet. *Dictionary compression for a scan-based, main-memory database system*. PhD thesis, Master thesis, Eidgenössische Technische Hochschule, 2009-2010, 2010.
- [5] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. *CIDR*, 2005.
- [6] L. Bouganim, F. Fabret, C. Mohan, et al. Dynamic query scheduling in data integration systems. *ICDE*, 2000.
- [7] R. Castro Fernandez, M. Migliavacca, et al. Integrating scale out and fault tolerance in stream processing using operator state management. *SIGMOD*, 2013.
- [8] G. Chatzopoulou et al. Query recommendations for interactive database exploration. *SSDBM*, 2009.
- [9] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. *SIGMOD*, 1999.
- [10] K. Church, B. Smyth, K. Bradley, et al. A large scale study of european mobile search behaviour. *MobileHCI*, 2008.
- [11] W. G. Cochran. *Sampling techniques*. John Wiley & Sons, 2007.
- [12] C. Cook. Database architecture: The storage engine, 2001.
- [13] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. *SIGMOD*, 2003.
- [14] M. C. Ferreira de Oliveira et al. From visual data exploration to visual data mining: A survey. *TVCG*, 2003.
- [15] M. N. Garofalakis and Y. E. Ioannidis. Parallel query scheduling and optimization with time-and space-shared resources. *SORT*, 1997.
- [16] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: killing one thousand queries with one stone. *VLDB*, 2012.
- [17] G. Giannikis et al. Crescendo. *SIGMOD*, 2010.
- [18] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. *VLDB*, 2014.
- [19] S. L. Greene, L. M. Gomez, and S. J. Devlin. A cognitive analysis of database query production. *HFES*, 1986.
- [20] J. Guo, X. Cheng, G. Xu, and H. Shen. A structured approach to query recommendation with social annotation data. In *CIKM*. ACM, 2010.
- [21] A. Gupta, S. Sudarshan, and S. Vishwanathan. Query scheduling in multi query optimization. *IDEAS*, 2001.
- [22] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. *SIGMOD*, 1999.
- [23] J. Hellerstein, R. Avnur, A. Chou, C. Hidber, et al. Interactive Data Analysis: The Control Project. *Computer*, 1999.
- [24] J. M. Hellerstein et al. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 2000.
- [25] S. Idreos. Database cracking: Towards auto-tuning database kernels. *CWI and University of Amsterdam*, 2010.
- [26] S. Idreos and E. Liarou. dbtouch: Analytics at your fingertips. *CIDR*, 2013.
- [27] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. *WWW*, 2009.
- [28] L. Jiang and A. Nandi. Snaptoquery: providing interactive feedback during exploratory query specification. *VLDB*, 2015.
- [29] M. Lai et al. System and procedure for concurrent database access by multiple user applications through shared connection processes, 1997. US Patent 5,596,745.
- [30] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. *ICDE*, 2013.
- [31] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *TVCG*, 2014.
- [32] Z. Liu, S. Natarajan, and Y. Chen. Query expansion based on clustered results. *VLDB*, 2011.
- [33] S. Madden, M. Shah, et al. Continuously adaptive continuous queries over streams. *SIGMOD*, 2002.
- [34] D. Maxwell and L. Azzopardi. Stuck in traffic: how temporal delays affect search behaviour. *IIIX*, 2014.
- [35] A. Nandi et al. Gestural query specification. *VLDB*, 2014.
- [36] G. Neumann and S. Schmeier. Exploratory search on the mobile web. In *ICAART (I)*, 2012.
- [37] H. Q. Ngo et al. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD*, 2014.
- [38] F. Olken and D. Rotem. Random sampling from databases: a survey. *Statistics and Computing*, 1995.
- [39] B.-H. Park, G. Ostrouchov, N. F. Samatova, et al. Reservoir-based random sampling with replacement from data stream. *SDM*, 2004.
- [40] M. Poess, B. Smith, et al. Tpc-ds, taking decision support benchmarking to the next level. *SIGMOD*, 2002.
- [41] A. Polonsky, R. Blake, J. Braun, et al. Neuronal activity in human primary visual cortex correlates with perception during binocular rivalry. *Nature neuroscience*, 2000.
- [42] P. Reisner. Human factors studies of database query languages: A survey and assessment. *CSUR*, 1981.
- [43] J. Rekimoto. Matrix: A realtime object identification and registration method for augmented reality. *CHI*, 1998.
- [44] J. Ren, W. Jiang, and C. Huo. Random sampling over streaming window joins. *ISDPE*, 2007.
- [45] B. Shneiderman. Response time and display rate in human performance with computers. *CSUR*, 1984.
- [46] B. Shneiderman, C. Williamson, et al. Dynamic queries: Database searching by direct manipulation. *SIGCHI*, 1992.
- [47] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. *VLDB*, 2004.
- [48] G. Takacs et al. Outdoors augmented reality on mobile phone using loxel-based visual feature organization. *MIR*, 2008.
- [49] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, et al. Hive-a petabyte scale data warehouse using hadoop. *ICDE*, 2010.
- [50] Q. T. Tran et al. Query by output. *SIGMOD*, 2009.
- [51] P. Unterbrunner, G. Alonso, and D. Kossmann. High availability, elasticity, and strong consistency for massively parallel scans over relational data. *VLDB*, 2014.
- [52] P. Unterbrunner, G. Giannikis, G. Alonso, et al. Predictable performance for unpredictable workloads. *VLDB*, 2009.
- [53] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. *VLDB*, 2001.
- [54] F. Xu, C. Jermaine, and A. Dobra. Confidence bounds for sampling-based group by estimates. *TODS*, 2008.