

Balancing Memory Load in D-CAPE Using Replication-Style Distributed Partitioning of Operator States

A Master's Thesis Proposal

By
Bradley A. Momberger

Thesis Advisor:
Prof. Elke A. Rundensteiner

Reader:

Department Chair:
Prof. Michael A. Gennert

Computer Science Department
Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609

April 6, 2005

Abstract

Distributed continuous stream query systems have been shown to meet the needs of a range of high-volume applications. However, in some applications the stream queries for these applications may require a wide range of sizes of allocated memory. In such cases it may not be feasible to continue to store the data for any of the largest operators on a single machine. In addition, for the benefit of the users of such a system, we would like to offer as much flexibility in query processing as possible. This includes making full use of resources on the network, even with a highly fluctuating volume and distribution of data.

This thesis proposal discusses an unexplored method of partitioning a query operator across several machines, without predetermining a destination for each new input for the operator. The proposed extension consists of the ability of the D-CAPE distributed continuous stream query system to activate a query plan operator on several machines simultaneously, and replicate new input data to each of these operators. Experiments will be conducted to observe the limits of high-volume query processing in D-CAPE with and without these new extensions, and to experimentally optimize these limits through the manipulation of several internal options.

1 Introduction

1.1 Motivation

Stream query systems [1] [8] [2] [3] build upon the knowledge gained from decades of research into databases to provide a framework for processing and monitoring data in real time [5] [6]. In a stream system, information is piped in from external sources and the system processes and outputs the results of a query over this data on-the-fly. In a stream query system such as D-CAPE [12] [13], the primary resources which must be managed effectively are CPU time, primary storage in terms of RAM, and network bandwidth. Of these resources, an overload on the CPU or network will cause the system to run slowly. However, when memory limits are exceeded in D-CAPE, the result is a crash of one or more query processors. In applications such as emergency healthcare monitors and combat-ready sensor networks, it is unacceptable to allow the system to fail under high memory load and lose in-transit data. Therefore, under these conditions memory is a critical resource that must be preserved at the expense of the others. In designing continuous stream query systems, one should seek to always keep data in main memory. The time interval required to run a responsive stream query system is too small to allow for moving queues and other held data to slower storage systems. Therefore, having multiple machines with large amounts of main memory may lead to queries being run efficiently over high-volume data sets.

Several proposed solutions to this problem use hash tables, a mechanism for efficient indexing and matching of data derived from research into the relational model. However, certain limitations are ingrained into managing distributed systems with hash tables. Parallel execution across machines is limited to hashable operators such as hash joins, and the hash buckets holding stateful data must be managed and occasionally rearranged.

It has been proposed that small states could be copied across machines to match against partitioned large operators. Updates to these small states would require a new input tuple to be passed to each machine with a partition of the operator state [7]. Intuitively, input tuples for states both small and large could all be sent in this same manner, though no research found to date seems to have explored this possibility.

1.2 Brief Overview of Approach

In this proposal I present *replication style* parallelization for the stream query system D-CAPE. Parallelization over a distributed query system allows query operators, especially stateful operators, to utilize the memory resources of two or more query processing nodes simultaneously. I will show that this will allow the system to continue to function normally under high-memory-usage conditions. As long as the aggregate total of available memory on the network is not exceeded, replication style parallelization will prevent system failure at the cost of a greater usage of network bandwidth. Even in cases where the total memory is exceeded, this method will delay the inevitable failure of the system long enough to extract significantly more output data. Using comparative experiments against the current implementation of D-CAPE, I will show that parallelization will increase the utility of the system when under a critically high load.

2 State of the Art

2.1 Review of D-CAPE

D-CAPE [12] [13] is a distributed stream query processing engine developed at WPI that supports constraint-aware, adaptive query handling over unbounded streams of data. Unlike similar systems such as XJoin [14] which deal with large but finite data sets, D-CAPE queries over potentially infinite streams of tuples. Pushing data to secondary storage is undesirable when processing a potentially infinite stream, due to the lack of a guarantee of time for data recall, unless the load on the system is known to have a high variance. A primary concern in the design of D-CAPE is accuracy; during execution, all output that should be returned from a query over D-CAPE will be returned. As a consequence, D-CAPE does not implement a load shedding algorithm like those seen in Aurora [1].

The D-CAPE system contains four major components: *query processors* which operate on the tuple streams, *stream generators* which feed base streams to the query processors, an *application* to receive output tuples from queries, and the *distribution manager* which gathers statistics and manages the distribution of query operators [13]. Each computer in the network is devoted to at least one of these tasks; therefore the term *machine* refers to one computer in the network, handling one or more of these four functions. The *query processor* is the most common task for a machine to run in D-CAPE, and also the most

resource-consuming.

As D-CAPE is a relational-model query system, its query processing capabilities take the form of relational *operators*. Operators in a stream context take a tuple stream as input and output a tuple stream with its mapping, filtering, or relational operation applied. A *query plan* in D-CAPE is a directed acyclic graph of operators where all leaves represent externally-supplied input streams, and all roots feed a processed stream to a receiving application.

Though operators in a stream context may not block, they may retain *states* for finite or infinite time. In a stream context, a state is a multiset of tuples held locally by an operator. For example, in a sliding-window relational join over two streams, the join operator will keep one state for each input stream to hold tuples from the respective streams. Tuples are discarded from these states when they become older than the time coverage of the window. It is easy to see that state sizes may grow large when data flow over the stream query system is heavy or windows are large.

2.2 Related Work

From the concept's inception, research into distribution of operator states for parallel processing [9] [5] has focused on sending tuples to localized target partitions using one or more hash functions. Hash joining and how it aids parallel processing is discussed extensively in [15]. However, *parallelism* conceived here by Wilschut & Apers was intended to pipeline and provide early output for multiway joins. These optimizations were originally tailored toward database systems running on individual servers. Until clustering of servers and shared-nothing architectures became prevalent, it was infeasible to distribute individual binary hash joins across multiple machines.

With the new prominence of distributed systems, the concept of using hash-based joins to aid in distributing joins across several machines has been realized at an even finer granularity by moving groups of hash buckets to separate machines. An example of this can be seen in implementations of the many-to-many producer/consumer relationship between query plan operators. The operators in the eXchange [6] package, and in its logical successor Flux [10] [11], are implementations of these relationships optimized for queries over memory-resident stream tuples. In a set of producer/consumer pairs, the output of each child operator is fed through the producer and hashed to determine which consumer it should be sent to. In this way, a stateful operator which can make use of a hash (such as an equality join) receives only tu-

ples that match the hash values for the state which that machine currently holds. In cases where hashing does not assist this process, such as with inequality joins, dynamic constraints, or quantified predicates, the intended direction of new tuples is unclear or undefined under these existing models. Thus usage of these hash-based techniques is limited.

IP multicasting [4] is an example of replication of data streams in such a way as to reach multiple interested hosts as efficiently as possible. In the traditional unicast model, there must be a single stream of data packets routed end-to-end between the source and destination, with no sharing of bandwidth even for the same packets. IP multicast allows routers to intelligently send packets to multiple recipients defined by a *host group* and represented by a single IP address. This only requires that the source send one stream to the host group address, and routing of the stream to each individual target is handled by routers along each path. Consider a similar structure in a data stream query processing context. Instead of having to replicate the entire base streams across several networks to increase output, replication of partially processed streams may be done at the operator level through routing operators. This allows operators on multiple machines to do the processing work that previously only one operator on one machine could accomplish.

The core concept of single-instruction, multiple-data (SIMD) architectures in CPUs and DSPs works on a concept of replication, where several similar data elements are pushed into parallel registers and a single instruction is replicated across all of them. Zhou & Ross showed in [16] that this technique may be applied to database systems, in order to cut loop execution time by a large fraction. The same result could be achieved by dividing the work of a large loop across several machines. If operator state partitions on different machines are assumed to be disjoint sets, then the effect of a loop iteration on each of several machines at once is equivalent to a loop iteration in a SIMD-aware DBMS.

3 Proposed Approach

Consider the example of a binary join operator in a stream query plan, joining two streams named *A* and *B*. Each tuple that arrives at the join operator from stream *A* is compared against the *state* for stream *B*, the tuples in stream *B* which have come in within a specified window of time or events.

Suppose the state for *B* grows very large and places a strain on the memory of its containing machine. We

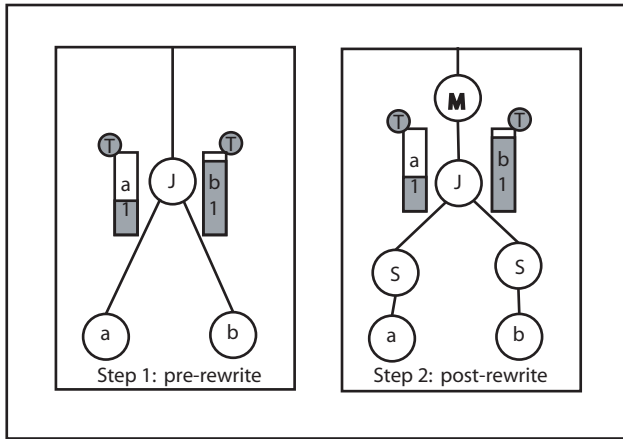


Figure 1: Query plan around a single join operator (J), rewritten to include split (S) and merge (M) operators

propose to *partition* the state for B across several machines; that is, have each of several machines hold a portion of the state for B . This partitioned state will behave just like the original single-machine state. This requires that new tuples from stream A will still be compared against all tuples; i.e., tuples in all of the partitions of the state for B . Also, the new tuple from A will be stored somewhere in the partitioned state for A .

Intuitively, if we want to compare a tuple against unstructured partitions across several machines, each machine will need to receive and review the tuple. To ensure that each machine receives each tuple, a *replicating* stream split operator will be inserted into the query plan as a child of the partitioned stateful operator. This replicating stream split operator will maintain a table of all machines which have a partition of the operator state, and feed a copy of each new tuple to each machine. Additionally, to ensure that the partitioned operator gives a consistent output, the streams which are output by the partitioned operators will be merged into a single stream using a dynamic union operator.

The main advantages of using this replicated approach are twofold. First and foremost, when using a replication style operator partitioning scheme, neither states nor state partitions ever need to be moved between machines. When a state partition becomes full, it needs only pass the responsibility of tuple storage to another machine with a less full state. Second, since replication does not rely on hashing to route new tuples, it is possible to use the partitioning scheme for a wider variety of operators than equality hash join. Some candidates for this scheme include inequality joins and specialized grouping functions.

3.1 Memory Management

Under the proposed partitioning scheme, query processors will be required to monitor their own *memory saturation* at all times, and feed the relevant statistics to the distribution manager. The term *memory saturation* refers to the availability of memory for a given state. If the checked state currently exists on the machine, memory saturation for the state measures the state’s ability to grow larger; higher saturation values indicate less ability. Similarly, memory saturation for a state which needs to be *created* on the machine is the portion of the machine’s memory unavailable to initially create this state; high saturation values indicate that a new state will put a strain on the machine’s memory. All memory saturation values are normalized to the interval $[0,1]$, zero for unsaturated and 1 for fully saturated. The measurement varies slightly between memory models for states. For example, in a pooled memory architecture, memory saturation for any query operator is the percent of the overall memory pool for queue tuples and state tuples which is filled.

3.1.1 Management and Allocation Schemes

The memory model implementation will handle the calculation of “saturation” for existing states and for the capability of a machine to create a new state as needed. Currently D-CAPE does not explicitly manage its memory, instead permitting the Java virtual machine to handle all memory and its allocation. Saturation is measured as total memory usage over the entire heap. When overall memory saturation exceeds “high” threshold, the largest states will need to be partitioned until memory usage growth has stalled. Other possible implementations include:

- a pooled model, described above in “Memory Management.” Split states which grow too fast, as they will be the greatest danger to system stability when memory resources are strained.
- an allocated model: fix state size at runtime, or initially allocate memory for states and later re-size as needed. Split states which get greedy.

3.1.2 Thresholds for Memory Saturation

In order to achieve our goal of using partitioning to increase system robustness, it is important to determine and use the optimal saturation values for “high” and “low” thresholds of memory saturation. Intuitively, these thresholds are dependent on the implementation of the memory model. During the measurement

phase, the optimal thresholds under the supported memory management systems will be experimentally determined.

3.2 The Storage Token

A *token* is a permission or resource held by only one machine in a network of clones. For each state partition in replication style parallelization, the *storage token* is a toggle on which the storage of new tuples is predicated. Consider time to be measured relative to the data streams being fed through the operator tree. At any point along the data streams, the one token for each state in the query plan is held by only one machine. This means that at any point in the data stream, a tuple being fed into the partitioned operator will be passed to all participating machines, but stored only in the state partition which has been nominated to hold the token for that segment of the input stream.

3.3 Target Selection

Under periods of high memory saturation, the distribution manager will decide to shift the storage token for a state to a different machine. Intuitively, extra machines should not be added into operator partitioning when machines already involved are holding very small states. When this is not the case, new machines may be added if they can handle the extra load. If no new machines are available for participating in operator partitioning, then the participating machines will need to play a rapid round-robin game to prevent any one partition from becoming too large. The distribution manager conducts the following protocol to determine which machine will then receive the token.

3.3.1 Priority for Choosing a Machine to Host a State’s Storage Token

1. The distribution manager gives first priority to any machine which is already involved in sharing load for this state, but whose saturation has fallen below the “low” threshold.
2. Second priority is given to any machine which is not sharing the load for this state, but has a saturation level for adding a new, unfilled state below the “high” threshold.
3. Lowest priority is given to machines whose existing share of the state in question is below the “high” threshold; in this case the distribution manager chooses the machine with the lowest saturation for this state.

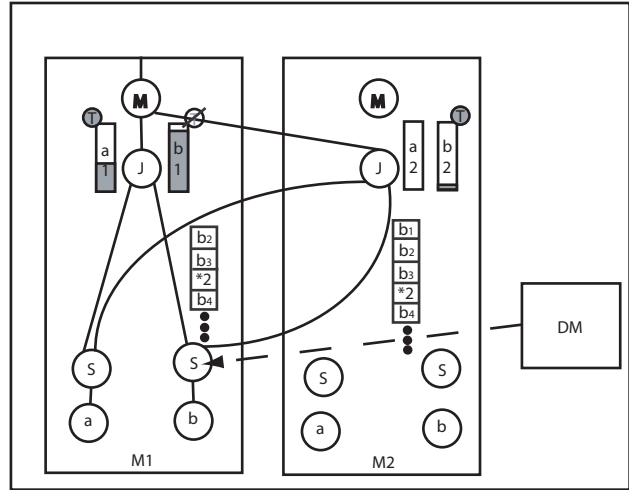


Figure 2: The distribution manager instructs the split operator above stream *b* to insert a command message (*2) into the join’s input stream. The join on Machine 1 (with the active split operator) will receive the command message first, thereupon giving up the token for the *b* state and ceasing to store new tuples there. Machine 2, the recipient of the token, receives the command message later, due to the latency of the network. However, the uniqueness of the storage token is preserved relative to the progress of the input stream. At this time, Machine 2 will give itself the *b* state token and begin to store new tuples in its *b* state.

4. Finally, if all machines are overloaded, the distribution manager will shift to a strategy which uses storage other than main memory, e.g., a ramdisk, RAID array, or storage server.

3.4 Operator Partitioning

When a query plan operator is running on only one machine, and exceeds the high threshold for memory usage for any of its states, this single-machine operator must be converted into a partitioned operator over several machines.

Because of the need to add operators to replicate single outputs and merge multiple inputs, query plans in D-CAPE need to be dynamically updateable to support this partitioning method. As Figure 1 shows, for any operator with partitioned states, all of its children in the query plan must be split operators (to send copies of tuples to the corresponding operator on each machine. See Figure 2 for an example of this data flow. The parent of this operator must be a merge operator to receive output from all operator partitions.

3.4.1 Query Plan Consistency Among Query Processors

While D-CAPE in its current form always passes all parts of a query plan to all query processors in the network, intuitively a query processor could function with only the knowledge of its active states and their outputs. This brings up the tempting idea of only adding split and merge operators to the machines that will use them. The expected best practice is to update every query plan with every addition/deletion, so that query plans remain homogeneous across all machines. While this is not the most network-efficient model, it may prevent the need for convoluted plan management code if other applications of operator addition are implemented.

3.5 Remission

When a machine has reached full saturation for any state and continues to receive new input tuples, the D-CAPE system is then *remiss* or in a *remiss state*. It can no longer be expected to provide complete answers to its queries. Experiments done on D-CAPE with partitioned operators will need to terminate when additional main memory can no longer be allocated. Therefore, handlers for failure states will be built into the D-CAPE system to check for events which signal remission.

A system which has encountered a program crash is remiss. Exception handling to save any needed data and signal failure to the network will wrap the main execution threads of the system. Additionally, for the purpose of experiments related to this project, a machine may be considered remiss when it is moving states or tuples from main memory to secondary storage. This includes any time that the virtual machine may see fit to extend its virtual memory into the system’s pagefile.

4 Evaluation

Since the goal of this project is to increase the robustness of the D-CAPE system, we will “stress test” D-CAPE under various conditions and configuration options. A suitable comparison metric for stress on a system lies in gauging overall performance prior to the occurrence of system failure or the activation of a stronger load-balancing method (since at this time, our method is insufficient).

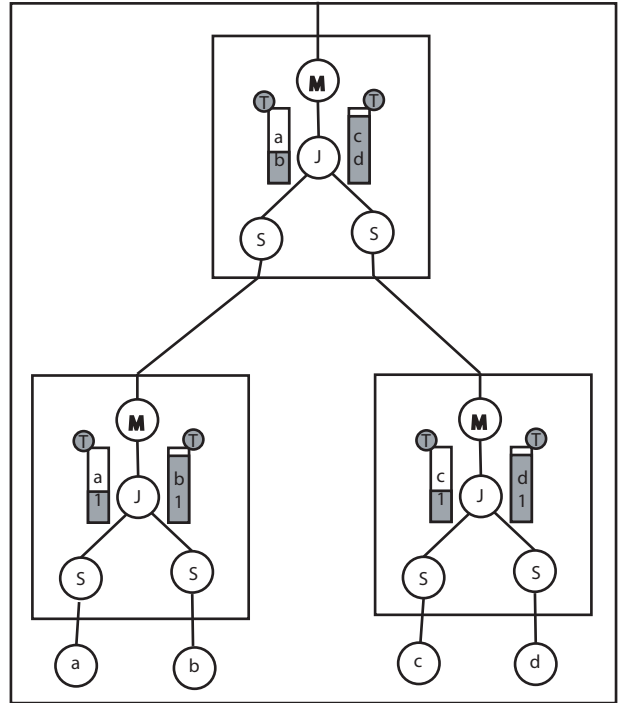


Figure 3: The encapsulation of split and merge operators around each stateful operator in a query plan. The dataflow between operators is the same as when each operator is active on only one machine.

4.1 Evaluation for Best Practices

The replication style parallelization implementation should first be optimized before a comparative analysis against existing work. The optimal “high” and “low” thresholds for memory saturation cannot be intuitively defined, and this is expected to majorly impact system performance. Therefore, a matrix of stress tests will be performed; each combination of “low” thresholds ranging from .1-.3 and “high” thresholds ranging from .75-.95 will be evaluated with the following process.

1. Test against three or more basic query plans (using only probabilistic selects and binary joins with 1:1 cardinality). The sequence of query plans represents a progression in the number of stateful operators, with the smallest query plan being just large enough to cause an eventual remiss system. All operators in these query plans will have a selectivity of .9 or greater to facilitate testing. For each query plan, we will determine the output before the first remiss state for each value of low threshold and high threshold in steps of .05.
2. Test against three or more query plans which in-

clude multiple joins of cardinality $[1,n]:[1,n]$, i.e. each tuple may join with multiple tuples from the other stream. Again, the query plans will have a progression in the number of stateful operators, and all stateless operators will have selectivity of .9 or greater. Joins and other stateful operators, by contrast, will have selectivity greater than 1.0. As in the first set of experiments, for each query plan we will determine the output before the first remiss state for each value of low threshold and high threshold in steps of .05.

In addition, tests will need to be repeated if D-CAPE is extended to use a managed memory model. The optimal threshold values will not necessarily be consistent across all possible memory models. Therefore, if managed memory is implemented, the above test battery will be repeated for each distinct memory model supported by D-CAPE.

4.2 Evaluation Against Current Models

Once the system has been optimized I will compare against existing configurations of D-CAPE for a better understanding of relative performance. The following test battery should provide a reasonably complete picture of the advantages and disadvantages of the partitioned operator method.

1. Repeat test (1) above as a comparative test against two different configurations of D-CAPE. Specifically, for each query plan we will compare the output before the first remiss state between D-CAPE with *partitioning disabled versus enabled*.
2. Using equality joins only, we will create a series of query plans representing a progression in the number of stateful operators, and with all operators having selectivity of .9 or greater. In this experiment, we will compare the output before the first remiss state between D-CAPE employing this *partitioning method* and D-CAPE employing a *hashing/partitioning method* such as the one implemented by [7].
3. Repeat test (2) from the best practices evaluation. As in the first comparative experiment set, we will compare the output before the first remiss state in D-CAPE with *partitioning disabled versus enabled*.

5 Pending Tasks and Time Schedule

- A. Verify whether the query tree operator structure already allows for more than one output, and whether this feature is stable. If this feature is deficient, fix or implement it. (1 week for completion)
- B. Add a split operator which takes advantage of the functionality outlined in (A). (1 week for completion)
- C. Derive “Merge” operator from existing relational union operator, while supporting any needed extra functionality such as a dynamic input array. (contemporary with (B))
- D. Implement token passing for operator states and, including recognizing the relevant command message and acting on whether a partition holds the token or not. (2 weeks)
- E. Implement statistics for “saturation of memory model” as percentage of memory allotted. (less than 1 week)
- F. Re-engineer D-CAPE’s handling of query plans to allow for runtime query plan rewrites at the query processor level. (4 weeks)
- G. Add a listener to the query processor event loop which checks for memory saturation above the threshold and informs the distribution manager. (1 week)
- H. Extend the D-CAPE distribution manager to initiate token passing and query plan rewriting. (1 week)
 - I. Allow for the distribution manager to gracefully remove states where they have emptied. (1 week)
 - J. Setup of experiment, including query plans and statistics collection framework. (2 weeks)
 - K. Best practices experimentation on HPC cluster. (3-4 weeks)
 - L. Comparative experimentation with other or no features. (3-4 weeks)
 - M. Analysis (2 weeks)

Writing, talks, and defense will be a continuous process throughout this period. For an additional 4-6 weeks after experimentation has completed there will be a period of thesis preparation and defense. The expected completion date is November 2005.

References

- [1] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.
- [2] S. Chandrasekaran. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR 2003*, 2003.
- [3] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaraqc: a scalable continuous query system for internet databases. In *SIGMOD '00*, pages 379–390. ACM Press, 2000.
- [4] S. E. Deering. Rfc1112: Host extensions for ip multicasting, August 1989.
- [5] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
- [6] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD '90*, pages 102–111. ACM Press, 1990.
- [7] B. Liu. Computing and maintaining integration views over distributed data sources, 2004.
- [8] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. In *CIDR '03*, 2003.
- [9] D. Ries and R. Epstein. Evaluation of distribution criteria for distributed database systems. Technical report, UC Berkeley, 1978.
- [10] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD '04*, pages 827–838. ACM Press, 2004.
- [11] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.
- [12] Timothy M. Sutherland. D-cape: A self-tuning continuous query plan distribution architecture. Master’s thesis, Worcester Polytechnic Institute, 2004.
- [13] Elke A. Rundensteiner Timothy Sutherland. D-cape: A self-tuning continuous query plan distribution architecture. Technical report, Worcester Polytechnic Institute, 2004.
- [14] T. Urhan and M. Franklin. Xjoin: A reactively scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 2000.
- [15] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.
- [16] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. In *SIGMOD '02*, pages 145–156, New York, NY, USA, 2002. ACM Press.