GREG GIBELING & NATHAN BURKHART
[GDGIB, BURKHART]@BERKELEY.EDU
UC BERKELEY
12/16/2005

Towards a Distributed Dataflow Database Platform

1.0 Introduction

Because they are often spread over large distances and have the potential for extremely complicated synchronization and parallelism constraints, distributed databases present an interesting application for dataflow languages and design techniques. Furthermore, when the thirty-year history of dataflow query execution (iterators) is taken into account, the pairing is nearly ideal.

This project started as a simple attempt to add database functionality to the P2 dataflow system, while taking full advantage of the fact that the overlay network and database could be specified in the same framework. Instead, we found that the tools and theoretical basis needed for unifying these disparate components do not yet exist, providing a second and more subtle goal for our research agenda.

In addition to documenting our implementation of some basic database operators, this paper describes the problems associated with unifying distributed systems design and database management within the P2 dataflow framework. In database terms, this document is a "checkpoint" on what has turned out to be a very promising vein of inquiry.

Section 2.0 Background gives our motivation for the short term goals of this project. Sections 3.0 Implementation and 4.0 Performance describe our implementation and its measured performance, leading to Section 5.0 Immediate Results, which analyzes the results of this implementation. Sections 6.0 Systems & Platforms and 7.0 Related Work discuss the overarching implications of our attempts, and tie these into heavily related, though seemingly distant, research. Section 8.0 Conclusions and Section 9.0 Future Work conclude the paper.

2.0 Background

In this section we discuss the motivation for our initial interest in combining distributed systems design and database management.

In recent years there has been an upsurge in interest in peer-to-peer networks and the applications they enable. Shady businesses like Napster aside, there is a tremendous potential for these systems; we would refer the reader to [1-4] for more information.

2.1 Chord and DHTs

Chord [4] represents one member of a growing class of overlay networks called Distributed Hash Tables (DHTs), which provide a simple hash table interface through which it stores data at logically distributed sites based on some globally known hash function. This provides a simple distributed storage abstraction which obviates the need for a broadcast lookup to locate the data.

We discuss Chord in particular, because it was the subject of the initial P2 research [2] that fed our work.

In Chord, computers, or nodes, are arranged in a ring using modulo-2 arithmetic, wherein each node knows the location of its neighbor in the ring and is responsible for all data that maps to a hash key between its own and that of the next node. To avoid using *O(n)* time to transmit a message, where *n* is the number of nodes, each node maintains a table of logarithmically distributed "fingers" pointing to various nodes around the ring.

Chord provides a very useful base layer for many distributed peer-to-peer services, and can even operate reliably under high network churn.

2.2 PIER

PIER [1, 5, 6] is a distributed query processor designed primarily to be run on peer-to-peer architectures ranging over potentially millions of nodes.

PIER relies heavily on DHTs, using them for query dissemination, indexing, tuple partitioning, operator implementation, and more. In the name of abstraction, the system is almost entirely unaware[1] of the underlying structure of the DHT, using only the simple hash table interface.

Because PIER has very little knowledge of the underlying network, it is unable to take advantage of any potential performance which could be gained by said knowledge.

2.3 P2

The P2 project [2] grew out of attempts [7] to build overlay networks using PIER and the dataflow ideas rediscovered[2] and expanded by the Click [8] team at MIT.

P2, with its specialized Overlog language, was designed specifically for constructing overlay networks from declarative definitions using continuous queries. However, under the hood, the result is a fairly straightforward dataflow system based on both the Click architecture and source code.

2.4 Open Interface

The goals of this project, as laid out in the initial proposal, were entirely focused on implementing database functionality using the existing elements in the P2 system. The idea was to explore possible network-aware database algorithms that would take advantage of the flexibility of having the database and overlay network specified in a single language: Overlog. In essence, we wanted to explore the simplicity and performance benefits which could be achieved by moving the PIER work into the P2 framework.

In particular, we felt it should be possible to clearly specify both the overlay network and database functionality, so as to allow us to explore various ways to map logical to physical schemas. We will discuss this in more detail in Section 3.5 Open Interface.

In the course of our work, however, we uncovered several other problems which must be addressed before this is possible. We will discuss this again in further detail, starting in Section 5.0 Immediate Results.

[1] PIER does exploit the multi-hop routing and callbacks used within the DHT to implement some dataflow operators, such as aggregation and join.

[2] There is a long history of dataflow architectures and languages; we will discuss this in Section 6.2 Dataflow.

3.0 Implementation

In this section we discuss, several major design decisions behind and obstacles to our implementation of aggregate and join operators in P2. We will confine ourselves to facts that directly affected our work, leaving analysis to Sections 5.0 Immediate Results and 6.0 Systems & Platforms.

In order to avoid locking ourselves into a single schema for the data in the database, our first major design decision was to implement this functionality in C++ using the P2 "elements" directly, rather than relying on Overlog. We discuss this issue in detail in Section 5.2 System Design and other possible solutions in Section 6.3 Languages, as it became a clear problem with our work.

3.1 Storage

The first step in our implementation was to add the ability to store data other than that required for Chord, to the test implementation of Chord provided by the P2 team.

Shown in the upper-right gray box in Figure 1[3], our core storage facilities consist of the ability to insert and query all of the tuples in any of a number of tables at a Chord node. This is carried out by sending the appropriate "`insertData`" or "`queryData`" tuples to that node.

This introduces the second major design decision: using tuples flowing through the Chord network to represent queries as well as data.

3.2 Broadcast

In order to support queries over all nodes, instead of just the one to which the "`queryData`" tuple is sent, the second feature implemented was a broadcast facility.

In order to make this feature fully general, our third major design decision was to take advantage of the ability of the C++ P2 backend to embed tuples as values within other tuples. Using this functionality, we simply send a tuple of the form `<broadcast,` *`dest`*`,` *`src`*`,` `<`*`data`*`>>` around the Chord ring. Upon receipt, each node sends the tuple to its predecessor, also extracting the `<`*`data`*`>` tuple and looping it back locally.

A simplified block diagram of the functionality required for this facility is shown in the "Broadcast" box at the top of Figure 1.

3.3 Distributed Aggregates

In order to quickly achieve some basic proof-of-concept functionality, we chose to implement distributed aggregation first. Trying to design the dataflow graph and P2 elements for aggregation highlighted a number of vital points regarding the specification of a distributed database, which are discussed in later sections.

Primarily, we desired to write the code for performing aggregations only once, independent of the data being aggregated. To this end, we were obliged to implement a new P2 element: the Reflection Aggregate. Our need for this element was one of the major reasons we worked in C++ rather than Overlog, because Overlog does not currently include the ability to work with custom elements or variable tuple types.

The reflection aggregate element, as shown in Figure 1, accepts tuples which describe the aggregate to be computed in the format `<aggregateData, …, groupKey, aggField, aggType>`: `groupKey`

[3] Figure 1 appears at the end of the paper, so as to remain readably large.

provides the equivalent of the GROUP BY clause in an SQL query, aggField is a list of the data fields over which aggregates should be computed, and aggType is a list of the aggregates to be performed, one per field in aggField.

To perform an aggregate over all of the data in the network, a client would simply broadcast an aggregateData tuple, with the result destination set to its own address.

3.4 Broadcast Join

As with our aggregate implementation, our join implementation was designed to provide both baseline results and explore the integration between the database and network overlay code.

We implemented two very simple join algorithms: a centralized join and a broadcast join. In the centralized join, the client simply queries the complete contents of both tables to be joined and then performs the join.

The broadcast join, on the other hand, uses the broadcast base functionality to broadcast the smaller of the two tables to all other nodes in the Chord ring. We had hoped to use this in conjunction with range partitioning, which is described in Section 3.5 Open Interface, in order to improve efficiency.

Again, in order to fully parameterize the join we used a Reflection Join element, which accepts tuples from both tables and a single "joinStart" tuple: <joinStart, …, LTable, RTable, LFields, RFields, joinTypes>. LFields, RFields, and joinTypes are three lists that specify the fields in the local and remote tables and the way in which to join them (equality and inequality are currently implemented).

3.5 Open Interface

Because the design of the overlay network is explicitly known, joins could be optimized by only involving the nodes at which relevant data is stored.

Furthermore, the partitioning and balancing of said data would provide a bound on the number of nodes involved with the join. To this end, we explored the possibility of range-based tuple clustering and load balancing.

Though it was never directly implemented, due to the obstacles mentioned in above sections, we have written pseudo-code based on the algorithms presented in [9, 10] that could be easily implemented in Overlog.

However, the complexity of this code as it would be written in the C++ P2 back end, in conjunction with our inability to overhaul the Overlog front end, kept us from a working implementation within the time frame of this project. This has turned out to be fortunate, as the conclusions and ideas presented in the last 5 sections of this paper are a direct result of these obstacles.

4.0 Performance

In this section we briefly present some basic performance numbers from our implementations.

4.1 Test Setup

Currently P2 has been adapted to compile and run on Fedora Core 3 Linux. Our test machine was an Intel

Pentium4, 3.0GHz with 256MB SDRAM and a four drive SATA RAID5 array attached to an Escalade 9500 controller.

We compared both centralized and distributed aggregate and join on Chord rings ranging from 2 to 12 nodes. At the high end of this range, we suspect our test machine began paging, or that there may be bugs in some of the asynchronous event-driven code, based on the severe drop in performance.

For test data, we inserted pseudo-random data into one table (two for join), with a random distribution and a cardinality proportional to the number of nodes in the system.

4.2 Distributed Aggregation

Figure 2 shows the cost in messages of the two methods of computing an aggregate, as a function of the number of nodes.

The curves labeled "central" represent an aggregate over the distributed data, which is computed at a central location. The curves labeled "distributed" represent the situation described in Section 3.3 Distributed Aggregates, where the aggregate is computed independently at each location.

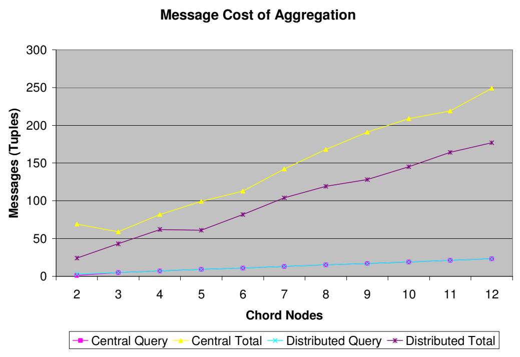

Figure 2: Message Cost of Aggregation

As expected, based on the linear increase in the cardinality of the input data, all of the curves in this graph approximate lines. Notice that the two "query" curves represent the cost of distributing just the query tuple, hence the perfect overlap.

4.3 Broadcast Join

In Figure 3, we present a comparison of two join algorithms, one which performs the join at a central location and one which performs the join in the Chord ring.

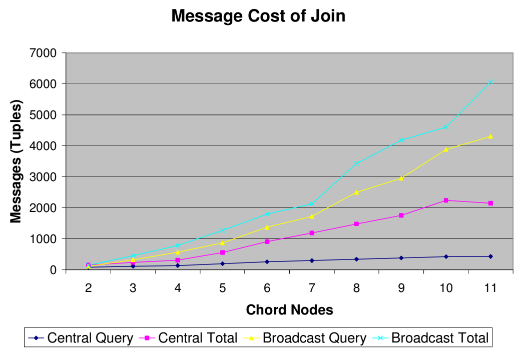

Figure 3: Message Cost of Join

As expected based on the increasing cardinality of the broadcast data, the messaging costs are polynomial in the number of nodes.

4.4 Result Bandwidth

Figure 4 shows a relatively pedestrian graph of result bandwidth as a function of the number of nodes for all four experiments. Again, the polynomial drop in bandwidth under the broadcast join is to be expected. We will discuss the implications of these numbers further in Section 5.3 Performance.

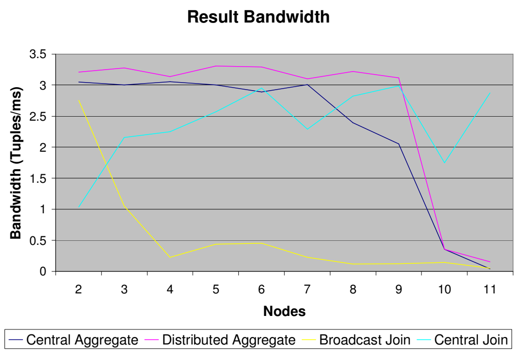

Figure 4: Result Bandwidth

5.0 Immediate Results

In this section we describe the short term results of our implementation work. We categorize our results by whether they represent an immediate success or a systemic problem that we were unable to overcome in the short term. Sections 6 through 9 suggest future research and long term solutions to these problems.

5.1 Successes

Despite our difficulty with integrating database functionality, we are currently able to support distributed data storage and retrieval on top of a Chord ring using the P2 dataflow system. In addition, we have baseline code on which to build future research.

At present, we can store data and run lookups, aggregate queries, and joins over it. The performance numbers in Section 4.0 Performance are not surprising or wonderful, but they match exactly what one would expect from these implementations.

The bottom line of this project is that, though we were easily able to add significant functionality to P2, thanks in part to the dataflow framework, there remain a number of major challenges before our a useful distributed database can be built.

5.2 System Design

Inarguably, the most significant problems we encountered during this project stemmed from the low-order nature of the Overlog language, including its inability to handle variable-length tuples or tuples which contain other tuples. This forced us to decide between the potentially major loss of flexibility incurred by using Overlog and the significantly increased development time and complexity due to writing in C++.

Ultimately, we decided to use C++ in the hopes that it would let us explore more issues by allowing flexibility in the schema of the data in our distributed database, rather than forcing it to be fixed within the same Overlog file as the Chord definition.

After actually attempting to implement these operators, we have come to appreciate that P2 and PIER are systems at opposite ends of a simple design decision: the level of integration of overlay network and database. PIER, by virtue of the hard DHT interface between the two, separates these components to the extent that significant performance gains will be unrealized due to its inability to exploit correlations between data, logical locations, and physical locations.

However, Overlog's current inability to separate the underlying overlay network from the database forces the programmer to either waste time working in C++ (though the P2 base code makes this much easier) or to couple the overlay network to the database so tightly that the two cannot easily be independently modified.

Our short-term solution to this limitation is embodied in the combination of the three major design decisions in Section 3.0 Implementation, and our creation of the "Reflection Aggregation" and "Reflection Join" elements. However, we feel that these are not valid long-term, general solutions. In Section 6.0 Systems & Platforms, we analyze this more completely, and provide possible solutions.

One of the original key points of this project was to explore algorithms and operators for overlay network-aware database query processing. While writing these operators in C++ is not totally impractical[4], the fact remains that it would be a long and arduous task to implement the many operators that have been suggested to date. Unfortunately, there are limitations imposed by the Overlog language itself that present significant obstacles to full-scale query processing. Potential improvements, such as the ability to include tuples within tuples or to deal with variable-length tuples, deserve further research. Issues such as semantics and typing rules become critical once these higher-level constructs become part of a language. See Section 6.3 Languages for a further discussion.

5.3 Performance

Our primary goal in presenting the performance graphs in the previous section was clearly not to demonstrate that our implementation is efficient, but rather to demonstrate that it performs exactly as expected. This is a sign that the P2 dataflow framework, even when used directly in C++, represents a significant step forward in the abstraction of both overlay network and database functionality.

However, the fact remains that thus far this abstraction has come at a heavy performance price. While the P2 group showed respectable results [2] for Chord, large scale databases have much more demanding performance targets: high end systems are beyond the 1MtpmC on the TPC-C benchmark, and even lower end systems often manage 10,000tpmC. During our tests, some queries took up to a second over a mere 200 data points. This means the current implementation is, generously, between 3 and 8 orders of magnitude slower than the top of the line. While we acknowledge that this is an unfair comparison, the fact is that with this performance disparity there remains no reason to consider this work for a production system. We offer possible solutions to this issue in Section 6.0 Systems & Platforms.

[4] Our implementations are on the order of 500 lines of C++, plus ~300 lines of test code.

6.0 Systems & Platforms

As described in Section 5.2 System Design, the P2 and PIER systems represent opposite ends of a design spectrum, where PIER uses an opaque interface between the underlying overlay network and database and P2 would ideally use Overlog, requiring the full integration of the two.

Our short term solution was to try and find a middle ground, and we settled on using the P2 C++ back end. This allowed us to leverage the useful dataflow framework without constraining us to those computational structures expressible in Overlog.

In this section we propose and outline a platform for studying this design space, allowing systems from both ends of that spectrum to be described in one framework and language. Our suggestions are drawn directly from the problems we encountered during this work, as outlined in above sections.

6.1 The Design Space

Given that PIER and P2 appear, in many ways, to be at opposite ends of the design space, the natural question arises: “What lies in the middle?” In fact, the most successful systems projects generally grow out of some practical compromise between extremes.

Based on the conceptual simplicity and natural mapping from traditional database iterators to a dataflow framework, systems like P2 provide a very natural way to explore this space. At this point, the problem becomes not, "How do we build a distributed database?" but, "How do we build distributed database systems?" The solution no longer lies in the P2 architecture, but in a combination of ideas from dataflow architectures, languages, and database systems research.

Fundamental questions about distributed systems remain hard to answer, in part because we lack a common high level framework and language with which to describe algorithms in such a way as to make them executable.

P2 and Overlog are examples of the inherent promise of this work. Despite our problems with Overlog, we at no point wish to contradict the conclusions of [2]. Furthermore, without the dataflow code base of P2, we could never have made significant progress on this problem in so short a time.

However, given a more powerful set of tools, as we will outline below, we might have met our original, even more ambitious goals.

In this section, we have outlined a case for a distributed dataflow database platform: a set of research tools for studying distributed database systems.

6.2 Dataflow

In this section, we provide a formal justification for the use of a dataflow framework. Beyond their inherent simplicity and natural match to existing database systems, dataflow systems offer a chance to separate the concepts of "synchronization" and "scheduling" of data, while managing the parallelism is a human understandable formalism.

Synchronization refers to the need for all inputs to be present before a computation can take place, such as the two inputs to an add element. Scheduling refers to the decision of when to actually perform that addition. Notice that the point of synchronization (when the second operand arrives) implies only the earliest point of scheduling.

In general, dataflow formalisms provide an effective way to express this style of "minimum requirements" for program execution.

Interestingly, even without the more formal dataflow language work, the structure of P2 actually suggests a few simple ways to parallelize the system overall: by simply partitioning a node into multiple dataflow graphs connected by explicit IPC, the various graphs can easily be executed in parallel without fear of adverse interactions. However, the primary justification for this split relies on the fact that dataflow frameworks are derived from message passing systems, and therefore generally eschew globally shared state,

Furthermore, there exists well-established research in dataflow languages and architectures, both of which have significant results that can easily be carried into this work. We touch on this again in Section 7.0 Related Work.

6.3 Languages

P2 is essentially a high-level interpreter for a first-order dataflow language. This has some serious implications for both performance and usability, the two major obstacles to our implementation work.

Low-order languages and programming systems typically severely complicate the expression of complicated algorithms and systems; simple examples can be drawn from classic type systems literature [11] and the relative complexity of a program written in assembly language versus Java.

In fact, we hypothesize that the success of Overlog is exactly due to the fact that it abstracts the details of the P2 elements: namely connections, push and pull dataflow, and storage, behind a simple, easily readable language. Overlog raises the level of language abstraction, reducing the unnecessary details inherent in an implementation, e.g. of Chord directly in C++.

This bears direction on our second major design decision, the representation of queries as tuples and our use of hierarchical tuples. By using a dataflow language with higher level constructs, we can capture all the benefits of these ideas without resorting to C++. In addition, any research platform must provide a way to specify new database operators. Operators in C++ may be efficient, especially when compared to those written in Overlog, but dataflow compiler and architecture research has much more to offer [12-14]. Of particular interest is the suggestion that a type system could reduce a higher order language to one simple enough to implement, even in hardware [15].

As we move forward, exact semantics and typing rules will be required in order to build any large system. The lack of these facilities at the C++ level significantly retarded our work, as did the textual overhead of working in C++. Ideally, a dataflow language should be exponentially more expressive for systems like this, allowing plug-in style components similar to those used in reconfigurable hardware systems like JHDL (see Figure 5 of [16]).

7.0 Related Work

In this section we tie the ideas of the previous section into several widely different fields of research. This section builds on 6.2 Dataflow by suggesting past dataflow work on which we can capitalize.

7.1 Related Systems Work

Both in theoretical terms and in terms of the actual implementation code, P2 is closely based on the Click Modular Router from MIT [8].

However, a number of recent hardware-centric projects have also adopted dataflow or process network frameworks (Lee shows the two to be very similar [17]). For example, the RAMP [18, 19] project, with which G. Gibeling is currently heavily involved, is attempting to design a dataflow-based framework and language (RDL). The original goal was to simulate multi-processor computer architectures; however, the full extent of RAMP/RDL's usage is currently unknown, as the language is still in flux.

In a related vein, the Liberty project [20] at Princeton has designed a language, compiler, and simulation platform for designing low level hardware in a style similar to RDL.

In addition, dataflow design patterns have surfaced in recent distributed systems such as Google's MapReduce [21], which we mention for three reasons: it is easily applicable to a wide class of problems, it allows massive parallelism, and it represents a small subset of the overall promise of dataflow systems.

For a recent survey of dataflow related research we refer the reader to [22].

7.2 Related Database Work

Clearly the most closely related database research has already been discussed at length throughout this paper. However, research projects such as Eddies [23], TelegraphCQ [24], and SteMs [25] will fit well into an expanded dataflow database platform.

8.0 Conclusions

In light of the number of obstacles encountered during this project, it should be clear that there is still an immense amount of work to be done.

While this project was only semi-successful at providing a working solution, it does provide a foundation for our future research and a sketch what that will be. In addition, our results suggest interesting ties to existing research in significantly disparate fields such as languages, compilers, theory, architecture, and digital circuits.

New tools will be required to permit databases, overlay networks, and systems issues to be studied within a common platform. This will allow detailed study of, among other things, the tradeoff involved in determining the level of abstraction between the overlay network and query processor.

Distributed systems represent a continuously open field of research of which distributed databases are a small fraction, despite the magnitude of the problems there alone. In this paper we have presented our attempts and ideas for advancing both fields.

9.0 Future Work

In addition to giving suggestions for the next step in our research, this section covers the questions and ideas which originally attracted our interest in this field, which remain unstudied.

Based on our attempts chronicled in this paper, we conclude that the next logical step is to begin design and implementation of a distributed dataflow database platform. P2 should be considered a proof of concept for this platform, as it has allowed us to get this far.

With the proper platform in place, the next step is to study partitioning data in a widely distributed system. This has major implications for performance, as described by Ganesan [9] and in Section 3.5 Open Interface.

We also know of no major attempts to study distributed metadata so far. Metadata is one of the big successes of the DBMS world, as it allows a plethora of optimizations at the query planning level.

In a high level language, compiler optimizations and query optimizer research converge on the same problem, suggesting that the two areas of research may cross-pollinate.

In fact, Eddies [23] and SteMs [25] could easily be realized in a dataflow framework, perhaps to the extent that in many non-deterministic languages such as [26] an Eddy can almost be written as a first class language primitive.

Storage and its permanence remain open questions in the context of distributed databases. Researchers often disagree on the utility of permanent storage, and we know of few proposals aiming to provide it.

10.0 References

1. Huebsch, R., et al., *Querying the Internet with PIER*. 2003. p. 1-12.
2. Loo, B.T., et al., *Implementing Declarative Overlays*. 2005: UC Berkeley. p. 1-16.
3. Rodriguez, A., et al. *MACEDON: methodology for automatically creating, evaluating, and designing overlay networks*. in *First Symposium on Networked Systems Design and Implementation (NSDI '04). San Francisco, CA*. 2004.
4. Stoica, I., et al. *Chord: a scalable peer-to-peer lookup service for Internet applications*. in *ACMSIGCOMM 2001 Conference. Applications, Technologies, Architectures, and Protocols for Computer Communications. San Diego, CA*. 2001.
5. Huebsch, R., et al., *The Architecture of PIER: an Internet-Scale Query Processor*. 2005. p. 1-16.
6. Harren, M., et al., *Complex Queries in DHT-based Peer-to-Peer Networks*. 2005. p. 1-6.
7. Loo, B.T., J.M. Hellerstein, and I. Stoica. *Customizable Routing with Declarative Queries*. in *Third Workshop on Hot Topics in Networks (HotNets-III)*. 2004.
8. Kohler, E., et al., *The Click modular router.* ACM Transactions on Computer Systems, 2000. **18**(3): p. 263-97.
9. Ganesan, P., M. Bawa, and H. Garcia-Molina. *Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems*. in *30th VLDB Conference*. 2004. Toronto, Canada: Stanford University.
10. Ganesan, P. and M. Bawa, *Distributed Balanced Tables: Not Making a Hash of it all*. 2003. p. 1-5.
11. Cardelli, L. *Type systems*. in *ACM 50th Anniversary Symposium: Perspectives in Computer Science. USA. 1996.* 1996.
12. Beck, M., R. Johnson, and K. Pingali, *From control flow to dataflow.* Journal of Parallel & Distributed Computing, 1991. **12**(2): p. 118-29.
13. Papadopoulos, G.M. and D.E. Culler. *Monsoon: an explicit token-store architecture*. in *Seattle, WA*. 1990.
14. Dennis, J.B. and D.P. Misunas. *A preliminary architecture for a basic data-flow processor*. in *Houston, TX*. 1975.
15. Buscemi, M.G. and V. Sassone. *High-level Petri nets as type theories in the join calculus*. in *Proceedings of ETAPS 2001 - European Joint Conference on Theory and Practice of Software (ETAPS). Genova, Italy. 2-6 April 2001*. 2001.
16. Bellows, P. and B. Hutchings. *JHDL-an HDL for reconfigurable systems*. in *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines. Napa Valley, CA*. 1998.
17. Lee, E.A. and T.M. Parks, *Dataflow process networks.* Proceedings of the IEEE, 1995. **83**(5): p. 773-801.
18. Gibeling, G., A. Schultz, and K. Asanovic, *RAMP Architecture & Description Language*. 2005, UC Berkeley.
19. Wawrzynek, J., et al., *RAMP Research Accelerator for Multiple Processors*. 2005.
20. Manish, V., N. Vachharajani, and D.I. August. *The Liberty structural specification language: a high-level modeling language for component reuse*. in *2004 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'04). Washington, DC*. 2004.
21. Dean, J. and G. Sanjay. *MapReduce: simplified data processing on large clusters*. in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI'04). San Francisco, CA*. 2004.
22. Gibeling, G., *The Art of Controlled Chaos: A Survey of Dataflow & Concurrent Programming*. 2005, UC Berkeley.
23. Avnur, R. and J.M. Hellerstein. *Eddies: continuously adaptive query processing*. in *2000 ACM SIGMOD. International Conference on Management of Data. Dallas, TX*. 2000.
24. Chandrasekaran, S., et al. *TelegraphCQ: Continuous Dataflow Processing for an Uncertain World*. in *CIDR*. 2003.
25. Vijayshankar, R., A. Deshpande, and J.M. Hellerstein. *Using state modules for adaptive query processing*. in

Proceedings 19th International Conference on Data Engineering. Bangalore, India. IEEE Comput. Soc. Tech. Committee on Data Eng. 5-8 March 2003. 2003.

26. Dijkstra, E.W., *Guarded commands, nondeterminacy and formal derivation of programs.* Communications of the ACM, 1975. **18**(8): p. 453-7.

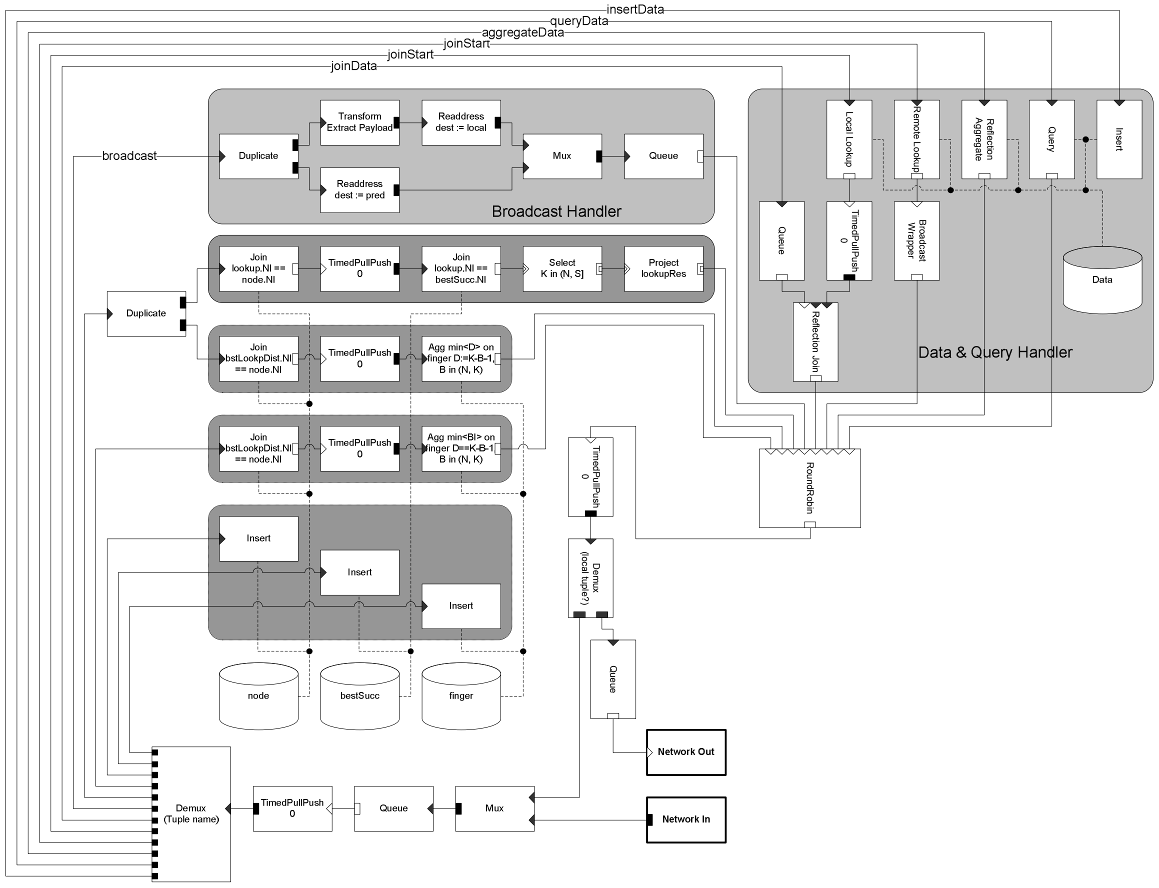

Figure 1: Database Operators and Simplified Chord in P2