A Paths Architecture for Caching and Prefetching Database Queries

Sirish Chandrasekaran, Samuel Madden
Computer Science Division
University of California, Berkeley
{sirish,madden}@cs.berkeley.edu

Abstract

This paper describes extensions made to the Ninja Paths Package to facilitate running database-style queries on the Ninja distributed architecture. These extensions took the form of enabling non-linear paths with multiple inputs and outputs, adding inter-operator queues, and switching to a message passing architecture for communication. Operators to do database joins and selects are described, and the CNQ architecture for treating web-sites as ODBC sources is discussed. A simple application which composes data from AC Transit and BART to generate travel-plans is presented. Finally, based on poor performance results from that application, an extensible query cache is presented. Performance results are given demonstrating the usefulness of prefetching and query decomposition schemes for a search-engine workload. Related work and future research directions are summarized.

1 Introduction

A theme common to current research in the traditionally separate fields of databases and operating systems or networks, is the composition of different services over the system area or the wide area, to provide a data path, providing an arbitrarily complex service from simpler units. The advantages of composed services are described in [2], the primary one being the independence of the different services that are composed. This feature leads to simplicity of individual services, a greater number of composed services that can be created with a given set of sub-services, and cleaner fault tolerance due to the possible replication of these sub-services.

In the operating systems and networks domain, these paths of composed services are primarily conceived as a linear sequence of services through which data successively gets transformed as it flows through. A typical example is the use of paths in ICEBERG [1] to connect various telephony devices which use different protocols. A sample demo for the project is the use of paths to listen to MP3 music served by the Ninja Jukebox [9] on a cell-phone. An example of the use of paths in the Ninja [2] project is a chat service which transparently connects any pair of users irrespective of the devices or languages that the two end-users are using. For example a user speaking Tamil on his cell-phone would be able to send data to a user typing English on his keyboard by a path consisting of a speech-to-text operator and a Tamil-English language translator. Communication in the opposite direction would take place using a set of operators which perform the transformation in the reverse direction to the one mentioned above.

While sequential paths include an important class of applications, the database perspective on the problem of dataflow through different operators, allows one to generalize the problem and conceive of paths as query plans over the wide-area. This allows us to apply the extensive research done in the database community on query optimization, and fault semantics.

A typical application in this space, of interest to the database community is the join of data from various web data sources to provide a richer search functionality on the web. For example, one could join data from the web-sites of various transit agencies in an area to determine a route between two locations, which might use any combination of the different transit services. Another example is the join of SAT scores in various high schools to data from a real estate service on the web to find correlations between the two sets of data.

While these composed services have a richer functionality than linearly joined services, the application of paths to execute queries over data sources on the web raises a number of interesting problems, such as caching to improve performance, and failure semantics and mechanisms and performance optimization in the presence of replicated or similar data sources.
The remainder of this paper is organized into 8 sections. Section 2 briefly describes the design and implementation of the first prototype of the Paths system, which supported only linear compositions of operators. The system is evaluated and the lessons that were learnt from the prototype are described. Section 3 describes the architecture of the second version of Paths, which includes support for generalised query plans and web data sources. It also discusses Cohera Net Query, a software used to wrap web-sites and export an ODBC interface to their data. Section 4 presents a sample application that was developed with the paths infrastructure. Its performance is evaluated. Section 5 discusses the cache architecture that was built over the remaining infrastructure to reduce the performance impact of web-site queries and data fetches over the wide area. This section discusses caching based on user-preferences, support for refresh of periodically updated web-data; prefetching schemes to improve the cache performance, and query decomposition to support semantic caching. Section 6 discusses the performance of the paths infrastructure for web-queries with the use of caches. Section 7 discusses related work in the different areas subsumed by the Paths system. Section 8 discusses ongoing and future work on this system. In particular, it discusses declarative and graphical approaches to create and manage paths. We conclude in Section 9.

2 Paths: Previous Implementation

The first prototype of Paths was built to demonstrate the feasibility of composition of services over the local area network. It supported query plans that were linear, and the typical application involved the transcoding of streaming data from one format to another as it flowed through various operators.

2.1 The First Prototype

Figure 1 shows the various modules of the original Paths Architecture and the supporting Ninja modules.

The subsystems of the Ninja Architecture with which the Paths system directly interacts are:

1. the Service Discovery Service [6], which helps discover services and their descriptions

Figure 1: Ninja Paths Architecture

(XML descriptions of the input and the output, which conform to a specific, well-known schema), and

2. XSet [12], a lightweight XML database used to store and retrieve the XML descriptions.

The modules specific to paths are:

1. the Automatic Path Creator, which generates a logical path, based on the end-user script. The logical path consists of the names of the operators and connectors between the endpoints. The Automatic Path Creator ensures that adjacent operators in the logical path have compatible input and output types as specified in their XML descriptions. It also ensures that the connectors provide the type of transmission semantics expected by the operators.

2. the Path Instantiator, which creates a physical path from the logical path, by locating/creating an instance of each operator in the logical path.

3. the Path Implementer, which starts a separate thread on each operator, and installs a connector stream to the previous and next operator in the path. The first operator in the path, called the source, can now begin writing data down the path with the expectation that the last operator, the sink, will receive that data in a format it understands.
In addition to these, the Paths system had a primitive fault tolerance mechanism to take care of fault in one or more of the operators in the path.

2.1.1 Limitations
The first prototype of the Paths system served as an excellent proof-of-concept implementation for the composition of services over the local area. However, there were a number of areas where it was lacking, either due to design/implementation decisions, or due to inherent problems in the solution approach:

1. Inability to support operators with fan-in or fan-out greater than 1: This was the primary flaw with the previous implementation. The prototype lacked a clear understanding of query-plans and as a result was unable to explore different alternatives to the execution of the query. It was also unable to take advantage of research in adaptive processing of queries.

2. Absence of support for messages: This was a implementation decision based on the typical use of Paths in ICEBERG. However, this limited the applicability of Paths in other domains.

3. Poor performance of path creation: In the first prototype, every new request for a path through an operator resulted in a separated thread being spawned for it. In addition to this, a new socket was opened for every individual path. The time taken to create a path therefore grew linearly with the number of operators in the path. For a path with 6 operators, the time to create the path was as high as 6 seconds.

4. Unsatisfactory fault-recovery semantics and mechanism: While the individual links in the path could have guaranteed communication semantics, there was no end-to-end concept of semantics.

5. Unsatisfactory typing of operators: While the choice of XML was driven by its high expressive power, the absence of a mechanism to enforce that typing negated the advantages of XML. Also, the problem of coming up with a single DTD to describe all possible operators is a hard one.

6. Absence of support for existing web-sites as operators.

2.1.2 Lessons Learned
Several important observations were derived from this first implementation:

1. Paths are compelling because they enable elegant solutions to a number of problems.

2. To enable support for database operations and a wider range of composed services, the system should support arbitrary fan-in and fan-out of operators.

3. Messages have to be supported for most applications of Paths on the web.

4. Existing web-sites are a huge source of data and services. The paths system should provide a mechanism to allow these web-sites to be part of a path.

5. Typing is a hard problem, given the tradeoff between expressive power and enforcability. The problem is orthogonal to the other issues involved in the building of this system. It can therefore be profitably ignored for the time being.

The next prototype of the Paths system was designed with these observations in mind. It is discussed in the next section.

3 Architecture

In this section, a brief discussion of the new features of the paths package is given. These features are complimentary to the features of the original package. We also present Cohera Net Query (CNQ) as a tool for wrapping web-sites as ODBC sources and discuss how CNQ sites are packaged as Paths Operators.

3.1 Enhancements to the Paths Package

A number of basic enhancements were made to the paths package to rectify the problems discussed above. The first and most significant change was support for non-linear paths. Operators can now have multiple input and output connections. To better support aggregation of multiple inputs, data-streams were replaced with message-queues: each operator has a single queue per input and is notified when messages arrive
on any queue. In response to an incoming message, an operator may choose one of several options:

1. Do nothing; simply absorb the message and invisibly adjust internal state
2. Send a reply message to the sender of the original message
3. Forware a new message down one of its outputs

Notice that this is a significant deviation from the data-flow model in the first implementation: operators are free to push messages backwards down a path and can also poll for input from multiple sources, combine those inputs, and produce a single result.

Operators implement a method, HandleMessage, which is called by the infrastructure whenever a message is available on some input. Messages are typed via the Java typing system – Operators use the instanceof intrinsic to determine the type of a message. Each message contains some basic information in addition to whatever their contents may be: primarily, the network address of the sending operator and the path id the message is travelling down. This information is filled in automatically before a message is sent out. Operator authors can implement new messages for their operators simply by extending this Message base class.

If an operator receives a message type it cannot handle, a default message handle sends and ErrorMessage back to the sending operator. The default handler ignores any error messages it may receive, so it is up to the operator author to write gracefully error handling code. There is currently no way to introspect on running operators and determine what message types they support; this is an important future extension if some automatic typing scheme is reintroduced into the paths implementation.

To reduce the performance overhead of creating a new path through an operator, the new version uses a single thread per operator and a single UDP socket per input or output. These resources are allocated when the thread is created, such that creating a path through an operator requires only adding the PathID for that path to a list of known ids. Operators can provide an initialization function to allocate storage for path specific state. When an incoming message arrives, its path ID is extracted and the operator is awakened with the appropriate storage container for that path. Although no thorough performance analysis of this system has been undertaken, results show path start up times on the order of tens of milliseconds once operators have been located via SDS. These enhancements corrected many of the concerns about the previous paths implementation and enable support for database-style operators, as discussed in the next section.

3.2 Database Operators

One way to describe a traditional database query plan is as a series of operators connected together via queues which share tuples (messages) via iterators. This description maps remarkably well onto a path: given some database style operators, an iterator interface, and a tuple-format, database queries can be run on the Ninja architecture trivially. We did this by implementing a simple Tuple and Iterator interface; these interfaces are very basic – the aim was to demonstrate a proof of concept database implementation running on top of paths. As such, they make no effort to be intelligent about the number of copies they perform or the way in which they allocate and reuse memory.

Given this tuple-iterator interface, operators that support selections and joins were implemented. The selection operator loops through a sequence of tuples, returning those which are filtered by a selection predicate which is passed in as a Java class. The Join operator is a naïve nested-loops-join which accepts two iterators and returns the result of the application of a join-predicate filter to every tuple in the cross product of the iterators. Currently join-predicates have been defined which support eqi-join and transitive closure across two similarly typed columns of the iterators.

One might question what the advantages of running database queries on the Ninja infrastructure are. After all, database engineers have been working very hard to develop the most efficient environment possible for running queries, so why try something new? It’s clear that the distributed nature of Paths and Ninja make them largely incapable of matching the performance of well-built database engine. But Paths do provide a number of benefits irrespective of performance: namely, operators are tolerant across faults, can be distributed on a cluster, and include advertisement and administrative tools. These benefits justified the engineering overhead of this implementation.

3.3 Supporting Web Queries via CNQ

With basic support for database operators in place, a good source of database tuples was needed.
In conjunction with the Telegraph group at Berkeley, an interface to the Cohera Net Query (CNQ) package was built. CNQ exports web pages and forms as ODBC datasources: a simple regular-expression language is used to describe how HTML formatted text maps into database tables. For example, the following lines describe the altavista search engine as a database table which supports queries of the form “SELECT * FROM altavista WHERE S=’UC Berkeley’”:

A wrapper class which packages ODBC tuples as Path tuples was written, thus allowing a number of truly interesting queries to be run on the paths infrastructure. In the next description, a simple demonstration application is described.

4 The TransitPlanner Demo

Data about Bay Area bus and rapid transit (BART) routes are currently available on the web. This data is made available in a basic tabular format, such as is found on paper schedules available and transit centers around town. No facility to query these schedules or determine a good route from one point to another is available. Finding routes is particularly hard for AC Transit busses – there are some 4300 stops on 100 different lines. How’s a naive commuter to get anywhere? Since tabular data is well suited to database tables, and the paths infrastructure supports database-queries over web sites, this seemed like an interesting demonstration application.

4.1 Features

The TransitPlanner is very simple: the user inputs a starting point and a desired destination, and the planner outputs all the routes from stops which string-match to the source to stops which string-match to the destination. The number of routes returned is pruned by eliminating routes which require the rider to transfer more than three times. Here’s a simple example query showing the path from Soda Hall at UC Berkeley to an author’s home:

Path from Hearst Ave and Euclid Ave to Arlington Ave and Thousand Oaks Blvd
Found result: Board line 52 at Hearst Ave and Euclid Ave
Transfer to 7 at Shattuck Ave and Cedar St
Depart at Arlington Ave and Thousand Oaks Blvd

4.2 Implementation

This package is implemented as a select from a CNQ operator which interfaces to the AC Transit web site for bus schedules. It is combined with locally stored data about BART routes. Queries are handled via a depth-first-search on a graph of known stops. The graph is periodically refreshed from the web-site on-line. The entire package was written in about four hours – the paths infrastructure combined with CNQ is a powerful tool for building useful applications.

4.3 Observations

Unfortunately, the performance of the demo is quite poor. It takes about thirty seconds to refresh the graph, as separate queries against all of the bus-routes, each of which is stored on a different page, are required. In general, web-overhead is the principal source of overhead in simple-queries: so much time is spent fetching remote, high latency data that it becomes the dominant factor in query time. Although the online nature of this demo makes the latency less obvious, a general solution is needed to promote the usefulness of web-based database applications. To that end, we implemented a general-purpose, extensible query cache, which is discussed in the next section.

5 Query Caching

The query cache sits in front of the CNQ wrapper and intercepts queries from Path Operators into the wrapper. It is implemented as a Ninja iSpace operator and as such exports an RMI interface for query processing.

Figure 2 shows the basic architecture of the cache. The next sections discuss each of these modules in more detail.

5.1 PathQueryCache

The PathQueryCache is the basic caching module. It caches exact matches on queries: only queries with exactly the same phrasing are considered equivalent. Queries are submitted via an RMI call to the CacheLookup method, and
results are returned via the paths iterator interface. The resulting iterator may be a wrapper for a CNQ result iterator, or may merely step through the cache’s internal tuple array.

The PathQueryCache is configurable in several ways:

1. Size: The size of the cache (in terms of number of cached results) can be set via an RMI call.

2. Replacement Policy: The replacement policy is user-loadable. New replacement policies simply override one method in the policy base class and can be passed in via an RMI call.

3. User preferences: Users can specify how long a particular data-source tuples should be cached for, or that a particular site should never be cache or will never expire. This data is stored in a local table, indexed by data source name.

Finally, the PathQueryCache will expire web pages based on expiration time from HTTP headers or the caching tags in the HTML header, unless the user preferences explicitly specify a page as non-expiring.

User preferences are an important part of caching, particularly in the web domain, as many pages which have expiring data (such as stock tickers) don’t use prescribed expiration methods. Since users will doubtless want up-to-date data for frequently changing information like stock quotes, the need a way to express this. User preferences are similarly important when cache or system features like query decomposition and alternate-site substitution come into play: the user may care very strongly about getting exact results from a particular site, or may not care at all.

5.2 Prefetcher

The prefetching module sits on top of the PathQueryCache. It runs an asynchronous thread which dynamically generates query requests based on the most recent queries which have been run and the time of day.

The query history portion of the prefetcher examines the most recent k queries. For each query in k, it determines the t queries which are most likely to follow. It adds these to a queue of queries to prefetch and then directs its thread to fetch those queries in succession. Each of the t queries for a particular k are fetched before the next query in k is processed. If the user runs a new query before the queue is emptied, then the queue is rebuilt with prefetching for the just-run query occurring immediately.

The time-of-day prefetcher partitions some period of time (day, week, month) into n time slices, with each slice representing the same amount of time. Any time query is run, its access count for the current time slice is incremented. Whenever a new time slice begins, the list of queries which have occurred during that time slice are prefetched, beginning with the queries most likely to occur during that time slice.

Because the prefetcher is potentially competing for valuable CNQ bandwidth with the user, it’s important that it allow user-queries to take precedence. Unfortunately, CNQ doesn’t provide a facility to abort a query in-progress. However, a simple scheme where the prefetching thread sleeps for a period of time proportional to the frequency of user queries helps to achieve this effect.

5.3 Query Decomposer

Since the queries that are being performed on the Cohera Net Query wrappers are SQL queries, we can employ more sophisticated methods than searching for an exact match in the cache. This is demonstrated by the work one semantic caching [8] in which a query is resolved into one part that is the output of other previous queries, and a
remainder which has to be fetched the usual way. For example, given a sequence of two queries:

1. SELECT *
   FROM STUDENT
   WHERE Age > 10 , and
2. SELECT *
   FROM STUDENT
   WHERE Age BETWEEN (5 and 15) ;

   part of the answer to the second query is contained in the result of the first query. Therefore, if we cache the results of the first query, we can immediately generate partial results from the cache while fetching the remainder part of the query from the remote data source.

   While query containment is easy to determine for certain classes of predicates, in the more general case, it reduces to the boolean satisfiability problem. As an example of a possible Query Decomposition scheme that can implemented on top of our extensible cache, we developed rules for resolving search-engine queries with AND, NOT, and OR predicates. The individual smaller predicates are then used to query the cache and if there is a miss, the results are returned from the search engine. These sub-results are then combined into the final result based on the semantics of the predicate. The intuition behind resolving a larger predicate into smaller ones is that it might increase the chance of a hit in the cache, allowing us to return partial results, if the semantics of the query permits us to do so.

   However, there is a caveat.

   1. Unlike traditional database systems where semantic caching techniques have been applied, the interposition of a Query Decomposer between the application and the cache, might change the answer that the application receives. That is, the query decomposer breaks the invariant that the answer is the same, whether or not you use the cache. For example, given the search string “Michael Jordan Basketball” to a search-engine which performs an OR condition on the search results of the individual terms to generate the result, but ranks the results by an AND condition, and outputs only the top ten results, the chances that all three keywords will occur among the top results is high. However suppose the Query Decomposer splits this query into three different queries for “Michael”, “Jordan” and “Basketball”. Now, if we perform a join on the top ten results for each of these queries, the chances that we will get a page with all the three keywords is lower than before, since the top ten hits each on “Michael”, “Jordan” and “Basketball” might not have anything to do with the basketball player.

   2. The query decomposer might also end up lowering performance if there are a number of cache misses, because there will be more accesses to the remote site, one for each sub-query.

   Despite these pitfalls, our measurements, which are described in a later section show that Query Decomposition usually improves performance significantly in the presence of correlated queries.

6 Performance

To evaluate the performance of this cache, a large and realistic workload was needed. The WebSpy query engine provides semi-real-time lists of queries which are being run against the WebCrawler search engine. We were able to extract a sample of about 4500 distinct queries from this site. For the purposes of the results presented here, we ran a sample of 1000 queries, 836 of which were unique. Because the WebSpy engine returns results in blocks of ten queries and sometimes returns the same block, many of the repeated queries were repeated in the exact same order as they originally occurred. As the results show, this makes the prefetching module appear to be very effective.

   As a general caveat to this performance analysis, it’s useful to realize that these results aren’t presented with the intention of producing a maximally efficient cache but rather with the goal of demonstrating that the cache presented here provides good performance in an easily extensible environment.

6.1 Cache Performance

Figure 3 shows the performance of the cache for the test workload, without and without prefetching at different cache sizes and with an LRU and a Random replacement policy. Since the number of repeated queries is 164, the maximum number of hits is 164.

   Notice that prefetching is extremely effective, even for very small cache sizes: at a cache size of ten, better than fifty-percent of the optimal result
is obtained. This is due to the regularity of the workload, as discussed previously.

Another interesting observation is that the Random policy becomes more effective than the LRU policy at a cache size of 400. This appears to be due to a peculiarity of the workload, such that increasing the cache size by 100 elements doesn’t allow anything new and useful to be placed in the cache in the LRU case, while random performance steadily increases at all caches sizes because the chances that a useful result is thrown out becomes smaller.

6.2 Query Decomposition Performance

Figure 4 shows the results of the same set of trials run with the query decomposer turned on. In this workload, there were about 650 distinct terms, so in about 350 queries one or more terms was repeated. The number of hits is the number of queries for which one or more of the subterms was already in the cache.

The query decomposer is thus very effective – in nearly a third of the queries, some data was in cache. As mentioned previously, however, a high hit rate does not mean that high quality results are returned. However, a hit means some result is available, and that result can be returned to the user which the CNQ wrapper goes and fetches the proper result.

6.3 Real World Results

To demonstrate the real-world effectiveness of the cache, a workload of 100 queries was run against an instrumented version of the cache and an uncached version of the code.

Figure 5 shows these numbers. RMI overhead accounts for a large portion of the time required to access the cache, but the overall per-call latency is still around 2ms. Contrast this with the CNQ wrapper – overall performance is between 400 and 700 ms, depending partly on how many of the queries are in Cohera’s internal cache (the performance of which we are unable to analyze).
7 Related Work

Paths spans a number of different areas, such as query processing, query caching, fuzzy data sources etc. There was been a lot of work in each of these areas, which is applicable to Paths.

1. Query Processing: Hellerstein et al. [3] describe adaptive query processing for online data sources which might have varying performance and delays. This is very relevant to paths for the kinds of applications that involve aggregation over large amounts of data from different sources. Such techniques however assume the commutativity and associativity of the operators in the path. Though this true for JOIN operators, it need not be true for more general applications where the semantic properties of the data provide the context for the execution of the service - for example, it is not legal to commute an English-Tamil translator and a Tamil-text-TamilSpeech operator.

2. Caching: Web caching has been extensively researched. Most of the work in networking has been focused on caching web objects such as images. They have no concept of storing results of a query.
Barnes et al. [5] describe an extensible caching mechanism and have devised a language CacheL [4] for manipulating this extensible cache.
Franklin et al. [8] have studied semantic caching, and describe how to determine overlap and differences between the space covered by two queries. Naughton [10] et al have also studied query containment in their work on semantic caching. They describe rules to determine containment of one query within another. Most approaches in this space work for limited classes of predicates such as range predicates, but the problem of query containment reduces to that of boolean satisfiability in the general case.
This problem is also very similar to that of materialized views.

3. Fuzzy Data Sources: Ronald Fagin [7] discusses the combination of data from multiple data sources which have different semantics in their output. In his paper he specifically discusses queries involving multimedia content where some of the predicates on a query have sorted results and others are sets. His solution can be generalized to other forms of fuzzy data. This has implications to the use of alternate query plans where the data sources are not replicas, but merely similar.

8 Future Work

While the current implementation improves upon the previous implementation of Paths, there are areas where there is rich future work to be done. The following is a list of some of the ongoing work and future work that we propose to do in this area:

1. **Declarative interface to paths**: SQL offers an excellent base for designing a language to create paths declaratively. There is work in progress as part of the Paths system towards extending SQL to be able to declare Paths. The two specific extensions are generalizations of the FROM and the WHERE clause of an SQL query:

   (a) **FUZZY FROM**: In the current SQL language, the FROM clause takes in table names, which are essentially specifications of schema. We extend the FROM clause to take in fuzzy descriptions of schema which might be satisfied by more than one table in the database (which could be the WWW). For example, we could have a query:
   ```sql
   SELECT *
   FROM §STOCKS;
   WHERE stock.name = INKT
   ```
   Here, stocks refers to a general type, rather than a specific instance of a table. Therefore, any stock server on the web, could be used to service this query. This is a useful feature, because often, it does not matter what the actual source is, as long as the data is obtained.

   (b) **TRANSITIVE WHERE**: To include the notion of Automatic Path Creation, we introduce a new operator that can be used in the where clause, called TRANSITIVE WHERE "¬¬". This operator takes two tables(or types) as input, and finds a path through the other tables in the database, such that any two consecutive tables in this sequence have a joinable attribute.
   With this feature, it is possible to make a query such as:
   ```sql
   SELECT Panini.out
   ```
FROM Aristotle, Panini
WHERE Aristotle.out -> Panini.in

To execute this query, it might be necessary to insert another table in the plan, Greek2Sanskrit which takes in a Greek word as input binding, and outputs a Sanskrit equivalent.

2. **Graphical interface to paths**: In addition to a declarative interface to paths, it would be useful to build a graphical interface that would allow one to create, monitor and dynamically modify paths.

3. **Parallel execution of alternative query plans**: With the use of the FUZZY FROM, it becomes possible to execute a query so that it executes over multiple similar or identical data sources in parallel. Thus, a path involving shopping data could be launched over multiple online shopping sites. This could be done for both performance and fault tolerance. Performance is achieved if the different sources return data in different orders so that more of the result space is achieved quickly. Fault tolerance can be achieved because failure of one of the data sources does not stop the execution of the query over alternate query plans. There are especially interesting problems in this space such as the problem of combining results from various plans that produce similar but not identical results.

4. **Determination of composability of schemas**: While the use of relational schema offers the advantage over SQL that it can enforce syntactical typing; it still does not help with the problem of determining whether two fields of the same type are semantically composable. To make Automatic Path Creation work over the WWW, there is future work towards defining a protocol which will allow exchange of information between operators to determine whether they are semantically composable.

9 Conclusion

The extensions presented to the original Ninja paths package provide a useful framework for exploring database-style workloads in the distributed system environment of Ninja. We have demonstrated the duality of Ninja paths and database query plans and shown an interesting demonstration application which was easily built using our infrastructure and Cohera web wrappers.

The extensible cache further improves the usefulness of the paths system by significantly increasing performance, particularly on queries which are decomposable or have very regular access patterns such that prefetching is effective. It extensibility is its biggest asset – since new cache modules are pluggable and it functions as a standard Ninja operator, service writers can write caching modules which understand the semantics of queries and the sites they reference in order to maximize performance.

We look forward to continuing to use this framework in the context of the Telegraph project to explore the possibilities of dynamic caching and ambiguous query formulation in the processing of web queries.

References


