Towards Declarative Queries on Adaptive Data Structures

Nicolas Bruno [1], Pablo Castro [2]

Microsoft, USA

[1]nicolasb@microsoft.com

[2]pablo.castro@microsoft.com

***Abstract*— In this work we look at combining emerging technologies in programming languages with traditional query processing techniques to provide support for efficient execution of declarative queries over adaptive data structures. We first explore available technologies such as Language-Integrated Query, or LINQ (which enables declarative queries in programming languages) and the ADO.NET DataSet classes (which provide various efficient alternatives to manipulate data in procedural terms). Unfortunately, combining the good features in both technologies is not straightforward, since LINQ over DataSets results by default in execution plans that do not exploit the specific characteristics of the data structures. To address this limitation, we introduce a lightweight optimizer that dynamically chooses appropriate execution strategies for declarative queries on DataSets based on their internal structure. To further enable declarative programming, we introduce a component that dynamically reorganizes the internal representation of DataSets, so that they automatically respond to workload changes. We experimentally showcase the features of our approach.**

I. Introduction

There is an increasing number of applications that use database systems (DBMS) as the ultimate persistent data store, but however need to manage large amounts of data outside of the DBMS. Stronger requirements for response time and flexibility, combined with the broad availability of commodity hardware with large main memory capacity, introduce new challenges to the way applications search and manipulate data in memory.

A sample typical scenario is analytical software for financial institutions. A common requirement in these environments is to obtain large amounts of data, typically through a combination of database access and other sources such as web services, and then allow a single user to explore various "what-if" simulations. The expectation is that the system is highly responsive to this user, even when patterns of use or application parameters change. Another scenario that has become increasingly common is the use of main-memory informal caches for web or multi-tier applications. In this case the middle-tier system loads some information from databases or web services into main memory, and then multiple sessions query the main-memory data instead of the database or web service. While this does not represent a fully-featured caching solution, it is a simple and effective strategy for cases where the goal is to lower the pressure on the DBMS servers by offloading some of the query executions over low-volatility data to the middle-tier systems.

Traditionally, application developers implement complex data structures and specialized logic to address the above scenarios. However, handcrafted solutions usually result in the following drawbacks: (i) the use of a non-uniform structure that makes it hard to support general query mechanisms, (ii) the use of a non-uniform set of algorithms to manipulate the data, and (iii) lack of opportunities for automatically adapting the lower-level representation of the data to the workloads and environmental changes outside of the control of the system and its administrators.

Recently, new technologies in the programming languages space started to address some of the drawbacks described above. For instance, standard rich data structures such as the ADO.NET `DataSet` classes provide a unifying set of APIs and a clear programming interface for adding and manipulating data imperatively (see Section II-B for details). However, `DataSets` lack a richer declarative query mechanism. Additionally, manually choosing the physical organization of the data structure to meet varying requirements over time is a difficult task that often cannot be done once and remain effective long term.

Another new technology is Language-Integrated Query, or `LINQ` for short (see Section II-A for details). `LINQ` enables software developers to write queries in a declarative way without imposing any specific execution strategy. `LINQ` also exposes extensibility mechanisms that allow to customize the way queries are executed against a particular data structure. However, the default implementation of `LINQ` operators is simplistic, which is only appropriate for manipulating small ad-hoc structures in memory. As the size of the in memory data grows, these naïve approaches to query processing become unacceptable from both flexibility and performance perspectives.

Our main observation is that the query formulation, optimization and execution techniques of traditional database systems become an interesting alternative to execute queries against native in-memory data structures. Specifically, these techniques can be leveraged to help building data manipulation infrastructures with effective declarative query processing capabilities. An extreme point in this direction would be to directly use a full-blown main-memory DBMS. However, this approach (i) does not provide the tighter language and environment integration that is possible using `LINQ` and `DataSets`, (ii) can significantly increase the overall footprint of the application, and (iii) introduces additional dependencies and complexity around application development and deployment.

The rest of the paper is structured as follows. In Section II we review both `LINQ` and the `DataSet` classes in the .NET framework. Then, in Section III we describe a lightweight optimization framework that selects good implementations for the declarative queries written in `LINQ` over `DataSets`. Next, in Section IV we introduce a self-tuning component that allows adaptation of the internal representation of a `DataSet` for dynamically varying workloads. Finally, in Section V we report an experimental evaluation of the techniques described in the paper.

II. Review of Existing Technology

We next briefly describe recent technology that extends programming languages with declarative query capabilities (Section II-A) and rich data types that provide efficient implementations for manipulating data (Section II-B).

A. Language-Integrated Query

Language-integrated query [1], [2], or `LINQ` for short, is a recent innovation in the programming languages space that introduces query-related constructs to mainstream programming languages such as C# and Visual Basic. `LINQ` is designed to work over any data source that supports a minimum "iterator-like" contract[1] (i.e., the classical *open/getNext/close* interface). Unlike embedded-SQL, query constructs are not processed by an external tool but instead are language first-class citizens.

As an example, the array data type exposes the iterator interface and therefore can be used as a `LINQ` source. The following C# code fragment initializes an array `A` and displays the double of all elements smaller than five:

```
int[] A = {1, 2, 3, 10, 20, 30};
var q = from x in A
        where x < 5
        select 2*x;
foreach (int i in q)
    Console.WriteLine(i);
```

In order to introduce query constructs into the programming language, `LINQ` defines "standard query operators", including all of the standard relational operations (e.g., projections, selections, joins), as extension methods to the iterator interface. When a query is formulated, a `LINQ`-enabled compiler mechanically translates the query operators into function calls without further analysis. The query above thus becomes:

```
var q = A.Where( x=>x<5 ).Select( x=>2*x );
```

which uses lambda functions. This intermediate representation is in turn translated into standard C# code as follows:

```
IEnumerable<int> q= Enumerable.Project(
                    Enumerable.Where(A, AF1),
                    AF2);
```

where `IEnumerable` is the base iterator interface, `Enumerable` encapsulates all query operators over IEnumerable classes, and `AF1` and `AF2` are anonymous functions generated automatically as follows:

```
bool AF1 (int x) { return x<5; }
int AF2 (int x) { return 2*x; }
```

The default implementation of the operators uses fixed, general purpose algorithms. For instance, the selection operator (`Where` above) is implemented by performing a sequential scan over the input and evaluating the selection predicate on each tuple. In turn, the `Join` operator uses a hash-based order-preserving implementation.

By formulating queries in `LINQ` we raise the level of abstraction on the query model with expressions composed of standard primitive operators. An interesting aspect of using `LINQ` is that the formulation of queries is now declarative. This property provides new opportunities to further enhance query performance by leveraging the independence between the query formulation and the strategy used to execute it.

Customizing `LINQ`*'s Execution Model*

A simple way to customize the execution of `LINQ` queries is by overloading the original implementations of the standard operators. For instance, if we replace the array `A` in the previous example with a structure that supports fast range queries (e.g., a binary tree), we would like to avoid iterating over all its elements and evaluating the `Where` predicate. We can achieve this goal by overloading the standard implementation of the `Where` operator with a specialized alternative that exploits the specific characteristics of the binary tree data type. This is a powerful mechanism that can considerably speed up query expressions. At the same time, however, it only allows *peephole-optimizations* that cannot take into account the global structure of the query. Depending on the specific data characteristics, we might want to sometimes do a global analysis before committing to some execution alternative. Unfortunately, for that purpose we need a global view of the query that is not available with the per-operator-overloading mechanism described so far.

To address this limitation, `LINQ` supports the concept of *expression trees*, which are in-memory data representations of query expressions that make the structure of the expression transparent and explicit. Rather than translating a `LINQ` query into function calls to the standard operators, we can alternatively instruct the compiler to transform the query into a abstract syntax tree that we can programmatically *manipulate* and *optimize* before executing. A crucial advantage of using expression trees in `LINQ` is that the query description and the specific implementation are not tied together. In this way, we can change how the query is internally executed without modifying how the query is expressed. We can even have alternative implementations of a given query and dynamically choose the most appropriate version depending on the context. This is a very powerful mechanism that we will exploit in the next section.

[1]In the .NET framework, the iterator functionality is encapsulated in the `IEnumerator<T>` interface.

B. Rich Data Structures

The Microsoft .NET Framework includes a large set of generic data structures such as dynamic arrays, hash tables and dictionaries. Conceptually, the ideas in the paper can be applied to all these data structures, but we focus on on a specific data structure, the `DataSet`, which is included in the ADO.NET library [3] and can be used to represent tables and relationships in memory[2]. Among the reasons for our choice, we note that `DataSets` natively support indexing capabilities, are conceptually more closely related to relational database concepts, and have been increasingly used in data-centric main-memory applications.

A `DataSet` is a container for `DataTable` and `DataRelation` objects. Each `DataTable` represents a table in the traditional sense, having columns with a name and a data type, and rows representing the actual data. `DataRelation` objects represent primary- to foreign-key relationships between tables, similar to the corresponding constraint mechanism in relational database systems. In addition to these basic elements, `DataSet` supports the definition of indexes on top of tables (called `DataViews` in ADO.NET terminology). A `DataView` is a filtered-index backed by a variation of red-black trees to provide efficient lookups and updates. Interactions between `DataSets` and external data sources (e.g., DBMSs) are enabled by special purpose `DataAdapters`, which handle data exchange based on specifications given by application developers.

Once data is loaded into a `DataSet`, all subsequent data manipulation occurs entirely in memory. `DataSets` expose various capabilities to find and manipulate data. For example, we can access individual rows of a `DataTable` based on their positions using array-like notation. Alternatively, we can create `DataViews` and use them to perform fast lookups. The code fragment below shows a possible implementation of the relational query $\Pi_{a,b}(R \bowtie_{x=y} S)$ for `DataTables` `R` and `S`, which iterates over all rows in `R` and for each row finds matches in `S`. If `S` does not contains an index on column `y`, it will be created automatically and used to find the relevant matches. This index is then kept as long as there are not updates to `S`.

```
foreach(DataRow rR in R.Rows) {
  DataRow[] rsS = S.Select(" x = " + rR["y"]);
  foreach(DataRow rS in rsS)
    Console.WriteLine(rR["a"], rS["b"]);
}
```

While the programming interface of the `DataSet` is fairly powerful and provides many options for searching and manipulating data, the overall interaction with data still happens in procedural terms. There is no way of declaratively describing what data is needed at a certain point in a program. Furthermore, since the basic operations supported by the `DataSet` and related classes are only primitive actions, most programs will have to choose and manually implement (usually re-inventing) the algorithms for each supported scenario. For example, we could have used a hash-join alternative to implement the code fragment above by creating a hash table on `R` and probe elements from `S`.

In the next section, we show how we can combine the declarative nature of `LINQ` queries with the available optimized implementations of `DataSet`, resulting in a higher level of abstraction without sacrificing efficiency.

III. `LINQ` on Rich Data Structures

Enabling `LINQ` to work over `DataSets` is straightforward. We only need to extend the specification of `DataTables` so that it support the iterator interface, which is very simple[3]. Then, we can write the join query above as follows:

```
from r in R.AsEnumerable()
join s in S.AsEnumerable()
    on r.Field<int>("x") equals
       s.Field<int>("y")
select new { a = r.Field<int>("a"),
             b = s.Field<int>("b") };
```

While this code fragment is much easier to write and understand, the default implementation does not take advantage of the specific characteristics of `DataSets`. Specifically, we are bound to a default join implementation even if existing indexes (`DataViews`) could enable more efficient alternatives. By using `LINQ` we gain a declarative framework for specifying queries over `DataSet`, as long as we are willing to pay the price in efficiency. Fortunately, `LINQ` expression trees provide the mechanism to take the best of both worlds.

To that end, we propose a run-time optimization phase to dynamically choose an execution strategy for queries executed against `DataTables`. Figure 1 shows the different phases involved in executing declarative queries on `DataSets` (containers in bold represent our proposed additions). Initially, the various `LINQ` expressions that make up the query are translated into standard C# code (see the introduction for an example), which is in turn compiled into intermediate language. At run time, this resulting code produces an expression tree that is then optimized. The optimization strategy is sensitive to the fact that data is in memory and thus the optimization phase can easily be a significant part of the overall execution time (to mitigate this problem, we keep the optimizer as lightweight as possible). Finally, the optimized expression tree is dynamically compiled into intermediate language which processes the main-memory `DataSets`. We also maintain runtime state to monitor and dynamically restructure the implementation of `DataSets` based on access patterns (see Section IV for details). We next describe the optimizer in terms of its search space, cost model and enumeration algorithm [5]. We note that the main contribution of this paper is not on new query processing techniques, but instead on a careful engineering of traditional database concepts in this new context.

[2]`DataSets` do not provide a specific mechanism for handling instances that do not fit in main memory and rely on the underlying operating system's paging capabilities for that purpose.

[3]In fact, this kind of support for `LINQ` over `DataSets` is expected to ship in the next release of Microsoft Visual Studio [4].

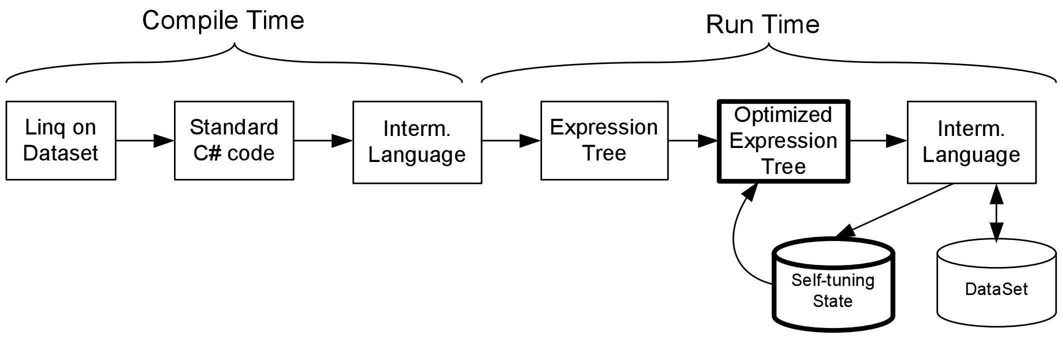

Fig. 1. Compile- and run-time phases on an implementation of our prototype.

A. Search Space

In addition to the default implementation of each operator provided by `LINQ`, we added new execution alternatives that typically rely on specific materialized indexes. For single-table selections scans, we added the possibility of using index-based execution strategies that speed up such process. Specifically, we identify the available indexes on sarg-able columns and consider an index-based execution plan that exploits available indexes. Consider, for example, a predicate $\sigma_{a=10 \wedge b+c<20}$. If a `DataView` on column a is available, we generate a plan that fetches all tuples in the index that satisfy $a = 10$ and then filters on the fly this intermediate result using $b+c < 20$. Clearly, if only a handful of tuples satisfy $a = 10$, this strategy is much more efficient than the full filter over the underlying table. For the case of equi-join predicates, we added *merge joins* and *index joins* to the default hash join alternative. Merge joins assume that both inputs are sorted by the join columns and is very efficient. Index joins scan the outer expression and use an index on the inner table to locate matches.

In general, there are several alternatives to implement each operator. Section III-B describes a simple cost model that allows us to pick the alternative with the least expected cost. Section III-C explains how we traverse the search space to obtain an execution plan.

B. Cost Model

Our cost model follows the traditional approach found in relational databases and is a function from plans (or sub-plans) into expected execution time. It relies on (i) a set of statistics maintained in `DataTables` for some of its columns, (ii) formulas to estimate selectivity of predicates and cardinality of sub-plans, and (iii) formulas to estimate the expected costs of query execution for every operator. These formulas exploit statistical properties of the input tables, knowledge about the specific algorithms implemented for each physical operator, and relative costs between the different primitive operations that are executed during query execution (e.g., memory accesses, or CPU instructions). We next describe these components in some detail.

Cardinality Estimation

Cardinality estimation returns an approximate number of rows that each operator in a query plan would output. We use standard techniques for this step that exploit statistics on the relevant table columns. To reduce the overhead, the statistical estimators that we rely on are very simple. Specifically, we use (if available) the maximum (*maxVal*), minimum (*minVal*), and number of distinct values (*dVal*) of each column. If statistics are unavailable, we rely on *magic numbers* until statistics are automatically created (see Sections III-B and IV-D).

We illustrate the cardinality estimation formulas using examples. The cardinality of base tables is obtained from the table metadata and it is exact. Consider a selection predicate $\sigma_p(T)$, where T is an arbitrary expression. The cardinality of the expression is in this case defined as $\mathit{Card}(\sigma_p(T)) = \mathit{sel}(p) \cdot \mathit{Card}(T)$, where $\mathit{sel}(p)$ is the selectivity of predicate p. The selectivity of a predicate depends on its structure, as illustrated below (where p, p_1 and p_2 denote predicates, c is a column, and c_0 and c_1 constants):

Predicate	Selectivity Estimation
$\mathit{sel}(p_1 \wedge p_2)$	$\mathit{sel}(p_1) \cdot \mathit{sel}(p_2)$
$\mathit{sel}(p_1 \vee p_2)$	$\mathit{sel}(p_1) + \mathit{sel}(p_2) - \mathit{sel}(p_1 \wedge p_2)$
$\mathit{sel}(c = c_0)$	$(\mathit{dVal(c)})^{-1}$
$\mathit{sel}(c_0 \leq c \leq c_1)$	$\frac{\min(maxVal(c), c1) - \max(minVal(c), c0)}{maxVal(c) - minVal(c)}$

Consider now a join predicate $T_1 \bowtie_{c_1=c_2} T_2$. The cardinality of the join expression is defined as:

$$\min(dVal(c_1), dVal(c_2)) \cdot \frac{Card(T_1)}{dVal(c_1)} \cdot \frac{Card(T_2)}{dVal(c_2)}$$

Cardinality estimation formulas for other operators are defined analogously.

Cost Estimation

We estimate the cost of an execution plan as the sum of the costs of each operator in the plan. These cost formulas are operator-specific and exploit cardinality estimates and knowledge about the internal algorithms. Additionally, we abstract the cost formulas in terms of certain cost parameters, such as the number of memory accesses, or the number of CPU operations. As a simple example, consider a scan operator over table T. We approximate the cost of such operator as:

$$\mathit{Card}(T) \cdot \texttt{MEM\_ACCESS\_COST}$$

where `MEM_ACCESS_COST` is the average cost of a memory access. As a more complex example, consider an index join

$R \bowtie_{x=y} T$, where R is an arbitrary sub-plan and T is the table with an index. Recall that the index join algorithm proceeds as follows. For each tuple read from the left sub-plan, it performs a lookup in the right-side index and finds the first matching tuple. Then, it starts traversing such index in order until the current tuple in the index does not match any longer. We model this procedure by the formula:

$$\begin{aligned} \mathit{Card}(R) \cdot \big(&\log(\mathit{Card}(T)) \cdot \texttt{MEM\_ACCESS\_COST}+ \\ &\mathit{dVal}(T)) \cdot \texttt{MEM\_ACCESS\_COST}+ \\ &k \cdot \mathit{dVal}(T) \cdot \texttt{CPU\_COST}\big) \end{aligned}$$

In other words, we sum $\mathit{Card}(R)$ times the cost of an index lookup plus the index traversal. The index lookup uses a logarithmic number of memory accesses to reach the leaf node in the index. It then traverses the index starting from the first match (an expected *dVal(T)* number of times), and each time it performs a memory access and some CPU computation to determine whether we keep obtaining matches. The remaining operators are similarly calculated.

In a one-time calibration phase, we carefully measured the actual execution times of the cost parameters (such as `MEM_ACCESS_COST` and `CPU_COST`) in a model machine to balance their relative weights in the cost functions.

Creating Statistics

As explained earlier, we need to provide estimates on the number of distinct values in a column, as well as the minimum and maximum values. If an index is available, we obtain these values exactly from the index metadata. Otherwise, the adaptive techniques of Section IV-D would automatically sample the appropriate `DataTables` to approximate these values balancing accuracy and efficiency.

Consider a table with N rows. To estimate the number of distinct values in a column we proceed as follows (see [6] for more details). First, we take a uniform sample of size n and define F_i as the number of elements in the sample that are repeated i times ($1 \leq i \leq n$). The estimation of the number of distinct values is then $\sqrt{N/n} \cdot F_1 + \sum_{i \geq 2} F_i$. The remaining unknown is the sample size n. We tried several alternatives and decided to use the value $n = 10 \cdot N^{0.55}$, which gives good results for the data sizes that we expect to encounter in practice.

C. Enumeration Architecture

The tight latency requirements in our main-memory scenario prevent us from using a sophisticated query enumeration architecture [7], [8]. In fact, it is not possible to spend a significant fraction of time optimizing queries because we risk compromising the overall query execution cost. Instead, we apply some initial heuristics to choose a good starting point, and then apply local, cost-based transformation rules to search for alternatives.

Our enumeration strategy consists of two passes. First, we perform a heuristic join reordering based on estimated cardinalities. Then, we perform a bottom-up traversal of the resulting logical tree, and use a local search at each operator. In other words, in the second pass we choose the best physical implementation for each operator, taking into account both the properties of the inputs and the availability of indexes. In this way, physical operator selection happens locally and does not consider changes across multiple nodes in the logical tree to obtain better physical plans.

As an example, suppose that the current node is a join operator. In this case, we consider 6 possible implementations: a hash join, a merge join, an index join, and the corresponding alternatives when swapping the inner and outer join inputs. While hash joins can always be applied, both merge joins and index joins require certain properties to be satisfied (i.e., the inputs of a merge join must be sorted in the join columns order, and index joins depend on the availability of an index in the inner-table join columns). We then use our model of Section III-B to approximate the cost of each alternative, and pick the one with the smallest cost.

IV. Self-Tuning Organization

The optimization framework described in the previous section certainly improves the overall performance of queries by dynamically choosing among a set of alternatives. However, as explained above, certain execution plans depend on specific `DataViews` (or indexes) being materialized. While this is not a problem if the application developer is certain about data and workload characteristics, it might become problematic to forecast in advance what indexes to build for optimum performance. Changes in the workload or data distributions only exacerbate this problem. In this section we describe a continuous monitoring/tuning component that addresses the challenge of choosing and building adequate indexes and statistics automatically, providing yet another abstraction to the application developer.

We automatically tune the set of indexes on the tables by adapting the techniques described in [9] to our main-memory scenario. A high-level description of our technique for automatically tuning indexes is as follows. As queries are optimized, we identify a good set of *candidate indexes* that would improve performance if they were available. Later, when the optimized queries are evaluated, we aggregate the relative benefits of both candidate and existing indexes. Based on this information, we periodically trigger index creations or deletions, taking into account storage constraints, overall utility of the resulting indexes, and the cost to creating and maintaining them. The key observation is that, by carefully deciding when to create and drop indexes, we can ensure that we do not suffer from late or wrong decisions and bound the error compared to a strategy that knows the future distributions of queries (see [9] for a formal analysis). In the rest of this section, we explain our approach in more detail.

A. Index Analysis

The query optimization procedure discussed in Section III searches a subset of the valid execution plans and returns the one with the minimum expected cost. At this point, we traverse the resulting execution plan and identify local sub-plans that could be improved if new indexes were materialized. Consider, as an example, the execution plan in Figure 2. The three sub-plans enclosed in dotted lines might be improved if suitable indexes were present. For instance, the selection predicate *name=“Pam”* over `DataTable` *Customers* can be improved if an index on *Customers(name)* is built (index I_1 in the figure). Analogously, both hash joins *might* be improved if indexes I_2 and I_3 are available, since we can transform the hash joins into index joins (note that index joins are not always more efficient than the corresponding hash joins).

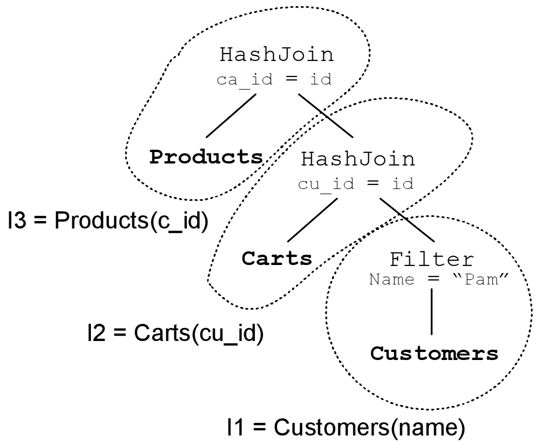

Fig. 2. Identifying sub-plans and candidate indexes.

For each sub-plan and candidate index that we identify in this way, we calculate the cost of the *best* sub-plan that exploits the index (for this purpose we reuse the cardinality values calculated during optimization and the cost model formulas). Additionally, for each sub-plan that already uses an index I_{used}, we calculate the cost of the *alternative* sub-plan that uses any existing index except I_{used} (i.e., we model what would happen if we dropped index I_{used}). For instance, consider index I_1 in Figure 2. In this case, we would calculate the cost of the *best* sub-plan that seeks index I_1 for all tuples that satisfy predicate *name=“Pam”*. Since the sub-plan does not use any existing index, we do not calculate the cost of an *alternative* sub-plan.

After optimizing a new query q we generate a *task-set* for q, which is a set of pairs $\{(I, \delta_I)\}$, where each I is either a candidate or an existing index, and δ_I is the amount that I would speed up query q (if I is a candidate index), or the amount that I's absence would slow down q (if I is an existing and used index). Since during optimization we know the cost c_p of each original sub-plan p, we can easily calculate δ_I for both a candidate index I (by subtracting the cost of the *best* plan that uses I from c_p), and an existing index I_{used} (by subtracting the cost of the *alternative* plan that does not uses I_{used} from c_p). We store the *task-set* of q along with the compiled execution plan for future reference. For update queries, we additionally store the expected number of updated rows and the columns that are updated.

B. Cost/Benefit Adjustments

When queries are executed, we leverage the preprocessing done during optimization and efficiently aggregate the potential benefits that we lose by not having the candidate indexes materialized, and the aggregated utility of existing indexes. Specifically, we maintain a value Δ_I associated with each index I[4]. When a query q is about to be executed, we retrieve q's *task-set* $\{(I, \delta_I)\}$. We then update Δ_I with $\Delta_I + \delta_I$ for each index I in the *task-set* of q. Additionally, if q is an update query, we subtract, from the Δ value of each index that should be updated due to q, the cost it would take to perform the index update. If we proceed in this way for each executed query, we can conceptually plot over time the value of Δ for any given –existing or candidate– index. We would then obtain a time series that grows if the index is useful (or would be if materialized), and gets smaller as the penalty of updating the index outweighs its benefits for query processing.

Figure 3 shows an example of a plot of Δ values over time for a given index. Intuitively, if the potential aggregated benefit of materializing a candidate index exceeds its creation cost, we should trigger a creation of such index, since we gathered enough evidence that the index is useful. In contrast, if the aggregated benefit of having an index oscillates but never increases beyond its creation cost, we can confidently avoid creating it, as the impact of such an index was not important enough. Analogously, if the aggregated benefit of an existing index decreases beyond the index creation cost, we should drop the index, because maintaining it has become too expensive. In [9] we show that this strategy results in a 3-competitive algorithm (i.e., it is no worse than 3 times the optimal algorithm that knows the future query distribution) for a restricted class of scenarios.

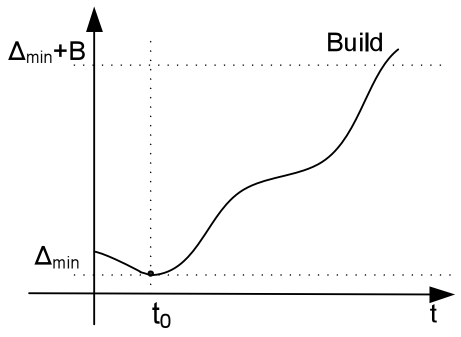

Fig. 3. Benefit of materializing a candidate index.

To determine whether a large enough increment or decrement beyond the index creation cost has taken place we can maintain the history of Δ values associated to each index. However, this option is too expensive (both in the space required to store Δ values and in the time it takes to process such time series). Instead, we effectively determine such conditions by maintaining the minimum and maximum

[4]Note that we maintain the metadata of candidate, non-materialized indexes in a global data structure. These candidate indexes do not help query processing but can eventually be created.

values of Δ (denoted Δ_{min} and Δ_{max}, respectively), and resetting the three counters to zero when an index is either created or dropped. In this way, we should create a candidate index I simply if $(\Delta - \Delta_{min})$ is greater than the cost to create it. Similarly, we should drop an existing index I if $(\Delta_{max} - \Delta)$ is greater than its creation cost. In general, if we use B_I to denote the cost to create index I, we define the *residual cost* of an existing index I as $B_I - (\Delta_{max} - \Delta)$. If $residual(I) \leq 0$, I should be dropped. Otherwise, $residual$ values indicate how much slack an index has before being deemed "droppable"[5]. Also, we define the *net-benefit* for a candidate index as $(\Delta - \Delta_{min}) - B_I$. If *net-benefit*$(I) \geq 0$, index I should be added. Also, positive *net-benefit* values indicate the excess in confidence for creating index I.

The procedure described above assumes that we can always create a new index. In practice, this assumption does not hold because there is a bound on the memory we can allocate for redundant data structures. In these situations, we need to (i) decide which indexes to create in case of competing alternatives, (ii) decide whether to drop a somewhat useful index to make space for better alternatives. In the following section we introduce the full tuning algorithm and address these challenges.

C. *Index Reorganization*

Figure 4 shows a pseudo-code of our algorithm for automatic index tuning. Each time a query is optimized, we generate its *task-set* T as explained in Section IV-A. When a query is executed, we retrieve its *task-set* T (line 1) and update Δ values for the indexes in T (lines 2-7). (We globally maintain in H the set of candidate indexes in the workload.) Note that lines 1-7 are very efficient because they only manipulate scalars.

Lines 8-14 analyze Δ values and optionally implement changes in the set of indexes that are materialized. First, in line 8 we drop all existing indexes I for which *residual*$(I) \leq 0$. In lines 9-14 we analyze the current candidate indexes and determine if we can create candidate indexes (and optionally drop existing indexes). For that purpose, we initialize ITM with all the candidate indexes I for which *netBenefit*$(I) \geq 0$, and process each index in ITM sequentially (we arbitrarily decide the order of elements in ITM to process, but several heuristics can be applied). When processing $I_M \in ITM$, we first obtain a subset of existing indexes ITD such that (i) I_M is still attractive if we subtract from its *netBenefit* the sum of residuals in ITD, and (ii) the space after removing ITD is enough to materialize I_M (note that ITD might be empty if we can create I_M without removing any existing index). In general, solving this sub-problem is computationally hard, and we approximate it by adapting the solution to the fractional knapsack problem. Specifically, we sort all existing indexes in $residual(I')/size(I')$ order, and progressively grow ITD until we obtain a solution or fail (this greedy approach is very efficient but, in some rare situations, can miss some solutions resulting in fewer alternatives to tune indexes). If line 11 returns a solution in ITD, we implement it in lines 12-14 by dropping all indexes in ITD (if any) and creating I_M.

[5]Note that the criterium for dropping indexes is based on a tradeoff between benefits and penalties rather than on when the index was originally created.

Addressing the Oscillation Problem

Although the algorithm described in the previous section is correct, it can lead to oscillations in some situations (i.e., cases on which multiple indexes are useful but there is space for only some of them). To understand this, suppose that we have a working set of useful indexes but do not fit in the available space. We know that, by definition, $residual(I)$ is bounded by B_I for existing indexes. At the same time, *net-benefit*(I) keeps growing for candidate indexes I as new queries arrive. Thus, eventually candidate indexes would replace existing indexes. But now, the indexes that we just dropped would start increasing their *net-benefit* values while the ones we just created would have a bounded $residual$ value. We are caught in an endless oscillation although the relative benefit of all indexes is similar.

To address this oscillation problem, we proceed as follows. Suppose that we are updating the Δ_I value of some existing index I in line 5 with an additional δ_I, but *residual*(I) is already B_I (i.e., the maximum value). After updating Δ_I to Δ_I+δ_I, $\Delta_{\max}$ would also be updated appropriately and *residual*(I) stays unchanged at B_I. To make this benefit explicit, in these situations we proportionally decrease Δ values of all the candidate indexes I' so that their new *net-benefit* value is $\max(0,$*net-benefit*$(I') - \delta)$. In other words, as existing indexes are helpful, we reduce the excess in confidence of the remaining candidate indexes by adjusting down their *net-benefit* values (but we never decrease *net-benefit* below zero in these situations).

D. *Statistics Management*

The cost model used to both choose execution plans during optimization and to obtain δ values for candidate indexes might benefit from the presence of statistics. However, we cannot greedily trigger statistics computation due to the additional overhead that this would impose. As a middle ground, we trigger statistics creation tasks on an index key column whenever $\Delta - \Delta_{min}$ is larger than a fraction (in our case, 0.5) of the index creation cost. In other words, after we gather some evidence about the usefulness of a given index, we create supporting statistics to have more accurate information in the near future.

V. Experimental Evaluation

We now present an experimental evaluation of the technology described in this work. For that purpose, we used a hypothetical online-store application with a product catalog organized by category, which allows users to browse the product catalog and to create one or more shopping carts where they can track the items they are interested in buying. Figure 5 shows the relevant portion of the database schema that includes the functionality described above.

```
IndexReorganization (q_i:query)
// Initially, global $H$=$\emptyset$ (no candidate indexes to materialize)
01 $T$ = get task-set for $q_i$

// Update $\Delta$ values
02 for each $(I, \delta_I)$ in $T$
03    if ($I$ is not materialized)
04       $H$=$H \cup \{I\}$;
05    $\Delta_I = \Delta_I + \delta_I$    // Section IV-B
06 if $q_i$ is update on table $T$ and columns $C$
07    $\Delta_{I'}$=$\Delta_{I'}$ - "update cost" for $I'$ on $T$ and $C$

// Remove bad indexes
08 drop existing $I$ if $residual(I) \leq 0$

// Analyze candidate indexes to materialize
// Approximate adapting fractional knapsack solution
09 $ITM$ = $\{I \in H : netBenefit(I) \geq 0\}$
10 for each index $I_M$ in $ITM$
11    $ITD$= subset of existing indexes such that:
         (i) $\sum_{I' \in ITD} size(I')$ fits in memory
         (ii)$\sum_{I' \in ITD} residual(I')+netBenefit(I_M) \geq 0$
12    if ($ITD$ is feasible solution)
13       drop $I' \in ITD$; create $I_M$;
14       $H$= $H - \{I_M\}; \Delta_{I_M} = 0$
```

Fig. 4. Online physical tuning algorithm.

Our application has three main workload types that appear simultaneously or independently at different times, in a way that is not controllable or predictable by the application or the administrators of the system. Specifically, the variance in workload characteristics mostly depends on the customer base and other external factors. The workloads are characterized as follows:

- *checkCarts($1) =*

```
        from p in Products.AsEnumerable()
        join cart in Carts.AsEnumerable()
          on p.Field<int>("id") equals
             cart.Field<int>("p_id")
        join c in Customers.AsEnumerable()
          on cart.Field<int>("cu_id") equals
             c.Field<int>("id")
        where c.name = $1
        select new { cart, p }
```

- *browseProducts($1) =*

```
        from p in Products.AsEnumerable()
        join c in Categories.AsEnumerable()
          on p.Field<int>("ca_id") equals
             c.Field<int>("id")
        where c.par_id = $1
        select p
```

- *updateProducts*, which consists of a large number of updates to adjust product information (e.g., seasonal price discounts).

An interesting aspect of this application scenario is that a large portion of the data (the whole product catalog) has very low volatility, making it a good candidate for caching. We used `DataSets` as middle-tier caches designed to offload work from the DBMSs. We generated 200,000 products, 50,000 customers, 1,000 categories, and 5,000 items in the shopping carts. To evaluate our techniques and highlight its features, we used a small workload W consisting of 20 instances of query *browseProducts*, followed by 20 instances of query *checkCarts*, followed by 10 instances of query *updateProducts*. To make the example interesting, we give enough memory to create only one index over the large *Products* table, and some on the remaining tables.

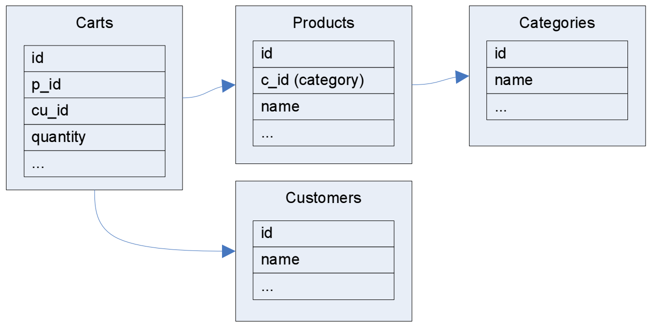

Fig. 5. Database schema for the experimental evaluation.

I_1	Categories(par_id)
I_2	Products(c_id)
I_3	Carts(cu_id)
I_4	Products(ca_id)
I_5	Customers(name)

Fig. 6. Indexes considered during the experiments.

Figure 8 shows a trace of the execution of workload W in our system. To keep Figure 8 concise, we refer by name to the indexes of Figure 6 and to the execution plans of Figure 7.

Initially, there are no indexes materialized. Consequently, the initial two executions of *browseProducts* use a hash join between *Products* and *Categories*, and a plain filter to select the desired categories (see plan P_1^1 in Figure 7). At this point, we gather statistics to obtain better cardinality estimations of both the filter and the join predicates. After another two instances of *browseProducts*, we create index I_1 on *Categories* and use it to seek the relevant category tuples (see plan P_1^2). After another execution of *browseProducts*, we create index I_2 on *Products*, which allows an index join alternative (plan P_1^3). The remaining 15 executions of *browseProducts* are evaluated in around 0.04 seconds, or around 22 times faster than the original plan. Next, we start evaluating *checkCarts* using the naïve plan P_2^1 that uses two hash joins. After 2 executions, we gather statistics and create index I_3 on *Carts*, which allows an index join between *Customers* and *Carts*. After four additional executions of *checkCarts*, we gather enough evidence on the benefit of index I_4 on product, and we choose to drop I_1 to make space (since the net benefit of I_4 exceeds the residual benefit of I_1). The remaining 12 executions of *checkCarts* use plan P_2^3, with two index joins and taking only 0.03 seconds. Finally, the updates to *Products* begin executing and the benefit of index I_4 gradually diminishes. After 3 executions of *updateProducts* we drop I_1, and the remaining 7 updates execute in a fraction of the original cost. We note that when we drop I_4, there is enough space to create index I_5, whose net benefit was positive but not big enough when I_4 was present. The total time, including index and statistics creation of the workload is 27.8 seconds. Figure 9 shows a trace of the execution of the same workload when the automatic index creation is turned off. We can see that the self-tuning alternative takes only 67% of the time.

Discussion

The experimental setting described above was designed to highlight the main self-tuning features of our techniques, but was rather simplistic in nature. In an actual deployment, workloads would typically be much larger and varied. We believe, however, that the experimental evaluation makes it possible to understand how those scenarios would be handled using our techniques.

For instance, suppose that we duplicate each of the *browse-Products* and *checkCarts* query instances 100 times. In this case, our techniques would process most of the queries very efficiently. Specifically, for this expanded workload, we would expect the naïve execution to last about 3,565 seconds (or around one hour) while the self-tuning evaluation would last for only around 34 seconds.

Of course, applications typically exhibit workloads with a mixture of queries at all times. Suppose that at any given time we have a stream of both *browseProducts* and *checkCarts* queries. If the allowed space is enough to accommodate all indexes, after a short period of time we would be able to evaluate all queries efficiently. In contrast, if we only have space for a single index on table *Products*, as in the evaluation above, we would create only one of the two indexes, which corresponds to the query that would make the larger difference in performance. If the relative frequency of queries varies over time, we would obtain something similar to the case of Figure 8, where we drop an index to accommodate a better one.

VI. Conclusions

We started this work with the goal of combining new technologies in programming languages and relational database systems to enable declarative query processing over adaptive data structures in the .NET framework. We first introduced various elements that we characterized as interesting for building data-centric applications. The choice of `DataSet` over ad-hoc data structures provided a uniform way of representing in-memory data and facilitated the use of generic algorithms for data manipulation. We then added `LINQ` to have access to a mechanism to declarative query formulation and also to unify the algorithms used to perform queries. With both uniformly-represented data and algorithms, we introduced a lightweight optimizer that adjusts the execution strategies using a simple cost-model for assessing query plans and sub-plans. Finally, we extended the query optimizer/execution engine with a continuous monitoring infrastructure that allowed the system to self-tune as workloads change.

We believe that adaptive data structures that can be accessed declaratively and also can adapt themselves to the workload coming from the environment without developer intervention will have an increasingly high applicability in current data-centric applications.

References

[1] "The LINQ project." accessible at `http://-msdn.microsoft.com/data/ref/linq`.

[2] E. Meijer, B. Beckman, and G. Bierman, "LINQ: Reconciling objects, relations and XML in the .NET framework," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2006.

[3] "ADO.NET," accessible at `http://-msdn2.microsoft.com/en-us/data/-aa937699.aspx`.

[4] "LINQ over DataSet," accessible at `http://-msdn.microsoft.com/data/ref/linq`.

[5] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the Symposium on Principles of Database Systems*, 1998.

[6] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya, "Towards estimation error guarantees for distinct values," in *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2000.

[7] G. Graefe, "The Cascades framework for query optimization," *Data Engineering Bulletin*, vol. 18, no. 3, 1995.

[8] P. G. Selinger *et al.*, "Access path selection in a relational database management system," in *Proceedings of the ACM International Conference on Management of Data*, 1979.

[9] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *Proceedings of the International Conference on Data Engineering (ICDE)*, 2007.

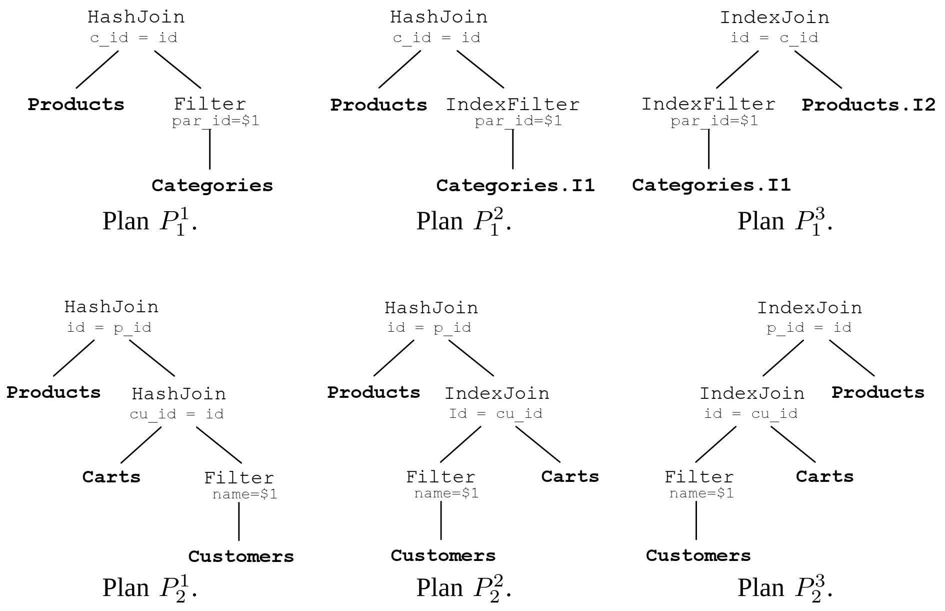

Fig. 7. Execution plans for the evaluation queries.

Query[Plan]	Background Action	Time per query (secs)
2× browseProducts [P_1^1]		0.9
	Create Stats(Categories.par_id)	0.01
	Create Stats(Products.c_id)	0.12
2× browseProducts [P_1^1]		0.9
	Create Index(I_1)	0.03
1× browseProducts [P_1^2]		0.89
	Create Index(I_2)	1.05
15× browseProducts [P_1^3]		0.04
2× checkCarts [P_2^1]		0.7
	Create Stats(Carts.cu_id)	0.01
	Create Stats(Products.c_id)	0.01
	Create Index(I_3)	0.13
6× checkCarts [P_2^2]		0.67
	Drop Index(I_1)	0
	Create Index(I_4)	0.64
12× checkCarts [P_2^3]		0.03
3× updateProducts		2.6
	Drop Index(I_4)	0
	Create Index(I_5)	0.16
7× updateProducts		0.56
	Total	27.8

Fig. 8. Generated schedule when tuning was enabled.

Query[Plan]	Time per query (secs)
20× browseProducts [P_1^1]	0.9
20× checkCarts [P_2^1]	0.7
10× updateProducts	0.56
Total	41.2

Fig. 9. Generated schedule when tuning was disabled.