A Comprehension-Based Approach for Query Optimization

Hao Fan, Gang Chen
International School of Software, Wuhan University, Hubei, P.R. China 430072.
Phone: +86 27 6877-8605, email: {hfan, g.chen}@iss.whu.edu.cn

***Abstract*—Comprehensions are a good query notation for database programming languages (DBPLs), which subsume query languages such as SQL and OQL in expressiveness. The AutoMed IQL is an intermediate query language based on comprehensions, which can be used to define the semantic relationships between schemas constructs. In this paper, we emphasize the query optimization problem for IQL queries. Since IQL can be smoothly integrated into several DBPLs, our approach is expected to be applied to semantic web environments.**

I. Introduction

In a database management system, there are many access plans to process a query and produce its answer. All query plans are equivalent if their final output is identical, but they are vary in their cost — the amount of time and space that they need to run is different from each other. This cost difference can be several orders of magnitude large [9]. A process of finding the plan needing the least cost for processing a query is called *query optimization*.

The issue of query optimization has been extensively discussed during the past several decades. [4] gives an overview of query optimization in relational systems. [2] discusses the utilization of materialised views for aggregate views' optimization in a data warehouse environment. [7] investigates the problem of nested query optimization and presents an approach, which revisits nested iteration plans for correlated queries and exploits the sort order of correlation bindings to produce plans that can be significantly faster than the corresponding set oriented plans. [8] indicates that extending query optimization into heterogeneous and non-relational data models is an open issue of parallel query optimization, and issues of query optimization for XML and over web services are discussed in [12] and [15], respectively.

AutoMed (see http://www.doc.ic.ac.uk/automed) is a heterogenous data transformation and integration system which offers the capability to handle virtual, materialised and indeed hybrid data integration across multiple data models. In AutoMed, the integration of schemas is specified as a sequence of primitive schema transformation steps, which incrementally add, delete or rename schema constructs, thereby transforming each source schema into the target schema. AutoMed uses a functional programming language based on comprehensions as its intermediate query language (IQL). IQL queries define the semantic relationships between schema constructs in each transformation step. In this paper we discuss an approach for optimizing IQL queries.

We firstly focus on Select-Projection-Join (SPJ) query optimization in relational algebra and describe a comprehension-based query optimization approach applied to the AutoMed IQL. Since comprehensions can be smoothly integrated into several DBPLs, which can subsume query languages such as SQL and OQL in expressiveness [16], [3], and precious work has also shown how relational, ER, OO, XML and flat-file data models can be defined in terms of the AutoMed low-level data model, the AutoMed IQL have the capability of representing queries in terms of multiple primary data models. Thus, our query optimization approach is also expected to be applied in semantic web environments.

II. Query Optimization

Query optimization refers to the process of producing a query execution plan representing an execution strategy for the query, which minimizes the cost of the execution. In different database environments, such as centralized and parallel, the considering details of query optimization are different. Parallel query optimization is a more complex problem in the way of considering the minimization of communication costs. However, in parallel query evaluation, a parallel query is translated into local queries, each of which is processed in a centralized way. In addition, the techniques for centralized query optimization can be extended into parallel query optimization. In this paper, we discuss SPJ query optimization without concerning communication costs.

Query Optimization Principles Optimization principles for SPJ queries are intending to decrease the cost of query evaluation by reducing the size of intermediate results or the number of times for iterating data collections:

1. Pushing forward select operators ahead of join operators, so that reducing the size of intermediate results;

2. Pushing forward projection operators ahead of join operators to eliminate attributes irrelevant to query results;

3. Combining several cascading of selections into one selection as following, so that reducing the number of times for iterating data collections;

$$\sigma_{\mathtt{C}_1}(\sigma_{\mathtt{C}_2}(...(\sigma_{\mathtt{C}_n}(\mathtt{D}))...)) \Rightarrow \sigma_{(\mathtt{C}_1;\mathtt{C}_n;...;\mathtt{C}_n)}(\mathtt{D})$$

4. Combining several cascading of projections into one projection by eliminating all but the final one as following, in which each $\mathtt{a}_i$ is a set of attributes of relation $\mathtt{D}$, and $\mathtt{a}_i \subseteq \mathtt{a}_{i+1}$ for $i = 1...n-1$: $\pi_{\mathtt{a}_1}(\pi_{\mathtt{a}_2}(...(\pi_{\mathtt{a}_n}(\mathtt{D}))...)) \Rightarrow \pi_{(\mathtt{a}_1)}(\mathtt{D})$;

5. Extracting common expressions in a query which need to be computed repeatedly, computing the result in advance

0-7695-3007-9/07 $25.00 © 2007 IEEE
DOI 10.1109/SKG.2007.9

and storing it during the query evaluation, so that eliminating redundant computing;

6. Indexing or ordering base collections in join expressions ahead of evaluation to increase the efficiency of join operators.

7. Finding the least cost evaluation plan for binary joins, i.e. $\mathtt{D}_1 \bowtie \mathtt{D}_2 \bowtie ... \bowtie \mathtt{D}_n$, in which the several inner collections are all base collections. This heuristic reduces the (potentially very large) number of alternative plans that must be considered.

Note that we do not consider the problem of selecting a best evaluating plan for binary joins in this paper. However, previous query optimization methods, e.g. System R Algorithm [14] and its extensions [10], can be referred to our approach.

In the rest of this paper, we describe a query optimization approach using the SPJ query optimization principles, mainly $1-5$, in terms of comprehension syntax, so that it can be applied to the AutoMed IQL.

III. The AutoMed IQL

A. The Syntax

We first introduce a subset of IQL, *Simple IQL* (SIQL). Supposing D, $\mathtt{D}_1$ ..., $\mathtt{D}_n$ denote bags of the appropriate type (base collections), SIQL supports the following queries: `group D` groups a bag of pairs D on their first component; `distinct D` removes duplicates from a bag; `f D` applies an aggregation function f (which may be max, min, count, sum or avg) to a bag; `gc f D` groups a bag D of pairs on their first component and applies an aggregation function f to the second component; ++ is the bag union operator, e.g. $\mathtt{D}_1$ ++ $\mathtt{D}_2$; $--$ is the bag *monus* operator [1], e.g. $\mathtt{D}_1 -- \mathtt{D}_2$; `map (lambda` $\overline{\mathtt{x}}$`.e) D` applies to each element of a collection D an anonymous function defined by a lambda abstraction `lambda` $\overline{\mathtt{x}}$`.e` and returns the resulting collection; finally, SIQL comprehensions are of three forms, $[\overline{\mathtt{x}}|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}_1; ...; \mathtt{C}_k]$, $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;$ `member` $\mathtt{D}_2$ $\overline{\mathtt{y}}]$, and $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;$ `not(member` $\mathtt{D}_2$ $\overline{\mathtt{y}})]$. Here, each $\overline{\mathtt{x}_1}$, ..., $\overline{\mathtt{x}_n}$ is either a single variable or a tuple of variables. $\overline{\mathtt{x}}$ is either a single variable or value, or a tuple of variables or values, and must include all of variables appearing in $\overline{\mathtt{x}_1}$, ..., $\overline{\mathtt{x}_n}$. Each $\mathtt{C}_1$, ..., $\mathtt{C}_k$ is a condition not referring to any base collection. Also, each variable appearing in $\overline{\mathtt{x}}$ and $\mathtt{C}_1$, ..., $\mathtt{C}_k$ must also appear in some $\overline{\mathtt{x}_i}$, and the variables in $\overline{\mathtt{y}}$ must appear in $\overline{\mathtt{x}}$.

The IQL syntax is defined in much the same way as SIQL is, unless collection expressions in SIQL are base collections while in IQL could be a nested IQL query. That is, more complex IQL queries can be encoded as a series of intermediate views defined by SIQL queries. Although illustrated within a particular query language syntax, our approach could also be applied to queries expressed in other query languages supporting operations on set, bag and list collections.

B. Comprehending SPJ Queries

Comprehension syntax can express the common algebraic operations on collection types such as sets, bags and lists [3] and such operations can be readily expressed in IQL and SIQL.

In particular, let us consider *selection* (σ), *projection*(π), *join* ($\bowtie$), and *aggregation* (α) (*union* ($\bigcup$) and *difference* ($-$) are directly supported in IQL and SIQL via the ++ and $--$ operators). The general form of a Select-Project-Join (SPJ) expression is $\pi_A(\sigma_C(D_1 \bowtie ... \bowtie D_n))$ and this can be expressed as follows in IQL comprehension syntax: $[\mathtt{A}|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}]$.

However, since in general the tuple of variables A may not contain all the variables appearing in $\overline{\mathtt{x}_1}$, ..., $\overline{\mathtt{x}_n}$ (as is required in SIQL), we can use the following two transformation steps to express a general SPJ expression in SIQL, where $\overline{\mathtt{x}}$ includes all of the variables appearing in $\overline{\mathtt{x}_1}, \ldots \overline{\mathtt{x}_n}$: `v1` $= [\overline{\mathtt{x}}|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}]$; and `v = map (lambda` $\overline{\mathtt{x}}$`.A) v1`.

The algebraic operator α applies an aggregation function to a collection and this functionality is captured by the gc operator in SIQL. E.g., supposing the scheme of a collection D is `D(A1,A2,A3)`, an expression $\alpha_{\mathtt{A2,f(A3)}}(\mathtt{D})$ is expressed in SIQL as: `v1 = map (lambda {x1,x2,x3}.{x2,x3}) D; v = gc f v1`.

C. Generating SPJ Queries

A SIQL expression can be easily transformed into common algebraic operations in terms of SPJ queries on collection types. In particular, forms of SIQL comprehension expressions are transformed as following:

$$[\overline{\mathtt{x}}|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \ldots; \overline{\mathtt{x}_n} \leftarrow \mathtt{D}_n; \mathtt{C}_1; ...; \mathtt{C}_k] \Rightarrow \sigma_{\mathtt{C}_1;...;\mathtt{C}_k}(\mathtt{D}_1 \bowtie ... \bowtie \mathtt{D}_n)$$

$$[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}}] \Rightarrow \sigma_{(\overline{\mathtt{y}}\ \mathtt{IN}\ \mathtt{D}_2)}(\mathtt{D}_1)$$

$$[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}_1;\ \mathtt{not}(\mathtt{member}\ \mathtt{D}_2\ \overline{\mathtt{y}})] \Rightarrow \sigma_{(\overline{\mathtt{y}}\ \mathtt{NOT\ IN}\ \mathtt{D}_2)}(\mathtt{D}_1)$$

SIQL query `map (lambda` $\overline{\mathtt{x}}$`.e) D` is transformed into $\pi_{\mathtt{e}}(\mathtt{D})$.

An IQL query is a SIQL query nesting other SIQL queries as its sub-queries, the transformation of the IQL query involves SIQL transformation rules recursively. For example, IQL query $[\mathtt{A}|\overline{\mathtt{x}_1} \leftarrow \mathtt{D}_1; \overline{\mathtt{x}_2} \leftarrow \mathtt{D}_2; \mathtt{D}_1.\mathtt{a}_1 = \mathtt{D}_2.\mathtt{a}_1; \mathtt{member}\ \mathtt{D}_3\ \mathtt{D}_2.\mathtt{a}_2]$ is transformed into $\pi_{\mathtt{A}}(\sigma_{(\mathtt{D}_1.\mathtt{a}_1=\mathtt{D}_2.\mathtt{a}_1;\mathtt{D}_2.\mathtt{a}_2\ \mathtt{IN}\ \mathtt{D}_3)}(\mathtt{D}_1 \bowtie \mathtt{D}_2))$.

IV. The Comprehension-based Query Optimization

In order to optimize a SPJ query expressed in relational algebra, we first transform it into an IQL query, and then decompose the IQL query into a sequence of SIQL queries. After that, the query optimization principles described in Section II are applied to the SIQL representations, and the obtained optimized queries can be either IQL or SIQL queries. Finally, we translate the optimized queries into local query expressions and send them to data sources to evaluate.

The comprehension-based query optimization consists of six phases: the *comprehension phase* transforms input SPJ queries into IQL queries based on comprehension syntax; *decomposition phase* decomposes IQL queries into sequences of SIQL queries; *optimization phase* applies query optimization principles to SIQL query sequences and generates optimized queries; *gathering phase* eliminates redundant parts of a query and reorganizes the query by gathering together the query parts which can be translated by the same data source, so that bigger sub-queries can be sent to each data source; *annotation phase* annotates queries by indicating which sub-queries have to be sent to which data sources; finally, *localization phase* translates an annotated sub-query into a local query expression and sends it to its data source to evaluate.

A. Comprehension phase

This phase is to comprehend a SPJ query within comprehension syntax. Section III describes that IQL and SIQL are both based on comprehensions, which have enough expressivity to comprehend SPJ queries in relational algebra. Also, references [3], [6] present how SQL and OQL queries can be functionally comprehended by comprehensions, whose methods are also considered in our approach.

Since we only consider SPJ queries in this paper, there are three primary translation rules for comprehending queries: A select expression, $\sigma_{\mathtt{C}}(\mathtt{E})$, is translated into: $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{E};\mathtt{C}]$; a projection expression, $\pi_{\mathtt{A}}(\mathtt{E})$, is translated into: `map lambda` $\overline{\mathtt{x}}$`.A E`; and a join expression, $\mathtt{E}_1 \bowtie \mathtt{E}_2 \bowtie ... \bowtie \mathtt{E}_n$, is translated into: $[\overline{\mathtt{x}}|\overline{\mathtt{x_1}} \leftarrow \mathtt{E}_1; \overline{\mathtt{x_2}} \leftarrow \mathtt{E}_2; ...; \overline{\mathtt{x_n}} \leftarrow \mathtt{E}_n]$.

Thus, only map and comprehension expressions appear in the generated IQL queries. In SPJ query transformations, above three rules are recursively involved. For example, SPJ expression $\pi_A(\sigma_C(D_1 \bowtie ... \bowtie D_n))$ is transformed into the following IQL query: `map lambda` $\overline{\mathtt{x}}$`.A` $[\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow [\overline{\mathtt{x}}|\overline{\mathtt{x_1}} \leftarrow \mathtt{D}_1; \overline{\mathtt{x_2}} \leftarrow \mathtt{D}_2; ...; \overline{\mathtt{x_n}} \leftarrow \mathtt{D}_n]; \mathtt{C}]$.

B. Decomposition phase

This phase is to decompose an IQL query into a sequence of SIQL queries. As described in Section III-A, IQL and SIQL queries are similar except that collection-valued expressions in IQL may be a sub-IQL query, while in SIQL must be a base collections or a variables defined by another SIQL query.

IQL queries are decomposed into sequences of SIQL queries by means of a depth-first traversal of IQL query trees. For example, suppose `v = D1 ++ [{x,z}|{x,y}` $\leftarrow$ `(D2 -- D3); z` $\leftarrow$ `[p|p` $\leftarrow$ `D4, member D5 p]; z < y]`, it is decomposed into: `v1 = D2 -- D3; v2 = [p|p` $\leftarrow$ `D4, member D5 p]; v3 = [{x,y,z}|{x,y}` $\leftarrow$ `v1; z` $\leftarrow$ `v2; z < y]; v4 = map (lambda {x,y,z}.{x,z}) v3; v = D1 ++ v4` .

C. Optimization phase

The optimization phase includes following six procedures respect to the optimization rules described in Section II.

1. **Eliminating map expressions** We consider map elimination rules in two scenarios: 1) a map followed another map, the two map can be merged. For example, `v1 = map lambda` $\overline{\mathtt{x}}$`.` $\overline{\mathtt{x1}}$ `E` and `v = map lambda` $\overline{\mathtt{x1}}$`.A v1` are merged into: `v = map lambda` $\overline{\mathtt{x}}$`.A E`; 2) a map followed a comprehension, the map can be eliminated. For example, `v1` $= [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{E};\mathtt{C}]$ and `v = map lambda` $\overline{\mathtt{x}}$`.A v1` are merged into: `v` $= [\mathtt{A}|\overline{\mathtt{x}} \leftarrow \mathtt{E};\mathtt{C}]$.

This phase is of pushing forward projection and combining cascading of projections.

2. **Pushing forward simple filters** Simple filters are filters involving variables, constants and comparison operators. The rule of pushing forward simple filters is that, for any simple filters appears in a single-source comprehension, i.e. $[\mathtt{A}|\overline{\mathtt{x}} \leftarrow \mathtt{E};\mathtt{C}]$, in which the source E is an intermediate definition (not base collection), the simple filter is pushed into the comprehension defining E. Since we have eliminated map expressions, the definition of E must be a comprehension.

For example, queries `v1` $= [\overline{\mathtt{x}}|\overline{\mathtt{x1}} \leftarrow \mathtt{E1}; \overline{\mathtt{x2}} \leftarrow \mathtt{E2}; \mathtt{C2}]$ and `v` $= [\mathtt{A}|\overline{\mathtt{x}} \leftarrow \mathtt{v1}; \mathtt{C1}]$ are translated into: `v1` $= [\overline{\mathtt{x}}|\overline{\mathtt{x1}} \leftarrow \mathtt{E1}; \overline{\mathtt{x2}} \leftarrow \mathtt{E2}; \mathtt{C2}; \mathtt{C1}]$ and `v` $= [\mathtt{A}|\overline{\mathtt{x}} \leftarrow \mathtt{v1}]$.

3. **Merging linear-connected comprehensions** Two comprehensions are *linear connected* means that, one comprehension contains a generator, i.e. $\overline{\mathtt{x1}} \leftarrow \mathtt{E1}$, whose source, E1, is an intermediate view, while the other comprehension is the definition of E1, whose generator sources are all **base collections**; in addition, there is no intersection of the generator sources of the two comprehensions. That is, if one base collection appears in a comprehension, it must not appear in the other.

If two comprehension are linear-connected, the definition comprehension can be merged into the other one. For example, the following two comprehension are linear-connected: `v1` $= [\overline{\mathtt{x1}}|\overline{\mathtt{xd2}} \leftarrow \mathtt{D2}; \overline{\mathtt{xd3}} \leftarrow \mathtt{D3}; \mathtt{C3}; \mathtt{C4}]$; `v` $= [\overline{\mathtt{x}}|\overline{\mathtt{xd1}} \leftarrow \mathtt{D1}; \overline{\mathtt{x1}} \leftarrow \mathtt{v1}; \mathtt{C1}; \mathtt{C2}]$, and the definition view v1 can be merged into v as: `v` $= [\overline{\mathtt{x}}|\overline{\mathtt{xd1}} \leftarrow \mathtt{D1}; \overline{\mathtt{xd2}} \leftarrow \mathtt{D2}; \overline{\mathtt{xd3}} \leftarrow \mathtt{D3}; \mathtt{C3}; \mathtt{C4}; \mathtt{C1}; \mathtt{C2}]$.

This procedure is to collapse multi-block queries into a single-block query.

4. **Assembling simple filters** In a comprehension, simple filters can be assembled with a generator to generate a sub-comprehension, if all variables appearing in the filters are derived form the head of the generator. I.e., supposes C1 is a filter and $\overline{\mathtt{x1}} \leftarrow \mathtt{E1}$ is a generator. If all variables in C1 appears in $\overline{\mathtt{x1}}$, a sub comprehension $[\overline{\mathtt{x1}}|\overline{\mathtt{x1}} \leftarrow \mathtt{E1}; \mathtt{C1}]$ is generated. We call C1 is an *internal filter* of E1.

For example, supposing C1 is an internal filter of E1 and C2 is of E2, comprehension $[\overline{\mathtt{x}}|\overline{\mathtt{x1}} \leftarrow \mathtt{E1}; \overline{\mathtt{x2}} \leftarrow \mathtt{E2}; \mathtt{C1}; \mathtt{C2}; \mathtt{C3}]$ can be translated into: $[\overline{\mathtt{x}}|\overline{\mathtt{x1}} \leftarrow [\overline{\mathtt{x1}}|\overline{\mathtt{x1}} \leftarrow \mathtt{E1}; \mathtt{C1}]; \overline{\mathtt{x2}} \leftarrow [\overline{\mathtt{x2}}|\overline{\mathtt{x2}} \leftarrow \mathtt{E1}; \mathtt{C2}]; \mathtt{C3}]$.

This procedure is to push forward select ahead of join.

5. **Handling member filters** The member filters reflect relationships of nesting queries. In our approach, we only consider optimizing comprehensions containing member filters in simple cases, that is, all generator sources of the comprehensions are base collections. In this case, we handle member filters as following.

Supposes `v` $= [\overline{\mathtt{x}}|\overline{\mathtt{x}} \leftarrow \mathtt{D}; \mathtt{member\ E\ x}]$, where D is a base collection. If E is a base collection, then it remains unchanged; if E is an intermediate view defined by a comprehension, then we consider it in two scenarios: (1) if data collection D does not appear in the comprehension defining E, which means v and E are unrelated but nested, then they remains unchanged; (2) if data collection D appears in the comprehension defining E, which means v and E are related and nested, then the definition of E can be merged into v.

For example, `v1` $= [\overline{\mathtt{y2}}|\overline{\mathtt{x1}} \leftarrow \mathtt{D1}; \overline{\mathtt{x2}} \leftarrow \mathtt{D2}; \mathtt{C}]$ and `v` $= [\mathtt{A}|\overline{\mathtt{x1}} \leftarrow \mathtt{D1}; \mathtt{member\ v1}\ \overline{\mathtt{y1}}]$ can be translated into: `v` $= [\mathtt{A}|\overline{\mathtt{x1}} \leftarrow \mathtt{D1}; \overline{\mathtt{x2}} \leftarrow \mathtt{D2}; \overline{\mathtt{y1}} = \overline{\mathtt{y2}}; \mathtt{C}]$.

6. **Pushing forward head patterns** Similar to eliminating map expressions, this procedure is to push forward projection and combine cascading of projections. A head pattern, such as A in comprehension `v` $= [\mathtt{A}|\overline{\mathtt{x1}} \leftarrow \mathtt{E1}; ...; \overline{\mathtt{xn}} \leftarrow \mathtt{En}; \mathtt{C}]$, can be pushed forward, if the comprehension contains intermediate views as generator sources.

For example, supposes generator $\overline{\texttt{xi}} \leftarrow \texttt{Ei}$ is a generator in v's definition and Ei is defined by $\texttt{Ei} = [\overline{\texttt{xi}}|\texttt{G1};...;\texttt{Gn};\texttt{Ci}]$. Without loss of generality, we assume that $\overline{\texttt{xi}} = \{\texttt{a1,a2,...,an,A.a1,A.a2,...,A.am}\}$, in which A.a1,A.a2, ...,A.am are the variables appearing in the head pattern A and filter C. Then, definitions of v and Ei are translated into: $\texttt{v} = [\texttt{A}|\overline{\texttt{x1}} \leftarrow \texttt{E1};...;\{\texttt{A.a1,A.a2,...,A.am}\} \leftarrow \texttt{Ei};...;\overline{\texttt{xn}} \leftarrow \texttt{En};\texttt{C}]$ and $\texttt{Ei} = [\{\texttt{A.a1,A.a2,...,A.am}\}|\texttt{G1};...;\texttt{Gn};\texttt{Ci}]$.

D. Gathering phase

After pushing forward head patterns, redundant queries might be generated, such as $\texttt{v} = [\texttt{A}|\texttt{A} \leftarrow \texttt{E1}]$, which can be eliminated from a query sequence. In addition, we reorganize a query by gathering together the query parts which can be translated by the same data source so that bigger sub-queries can be sent to each data source wrapper to evaluate.

For example, supposes $\texttt{v} = [\overline{\texttt{x}}|\overline{\texttt{x1}} \leftarrow \texttt{E1};\overline{\texttt{x2}} \leftarrow \texttt{E2};\overline{\texttt{x3}} \leftarrow \texttt{G3};\overline{\texttt{x4}} \leftarrow \texttt{G4};\texttt{C}]$, E1 and E2 denote collections derived from the same data source, as well as G1 and G2. In order to reduce the complexity of implementation, we only consider the case that expressions Ei and Gi are either base collections or single-source comprehension with the source being a base collection, such as $[\overline{\texttt{x1}}|\overline{\texttt{x1}} \leftarrow \texttt{D1};\texttt{c1}]$. Then, v can be redefined as following, and bigger sub-queries, i.e. v1 and v2, could be sent to its data source to evaluate, respectively: $\texttt{v1} = [\{\overline{\texttt{x1}},\overline{\texttt{x2}}\}|\overline{\texttt{x1}} \leftarrow \texttt{E1};\overline{\texttt{x2}} \leftarrow \texttt{E2}]$; $\texttt{v2} = [\{\overline{\texttt{x3}},\overline{\texttt{x4}}\}|\overline{\texttt{x3}} \leftarrow \texttt{G1};\overline{\texttt{x4}} \leftarrow \texttt{G2}]$; $\texttt{v} = [\overline{\texttt{x}}|\{\overline{\texttt{x1}},\overline{\texttt{x2}}\} \leftarrow \texttt{v1};\{\overline{\texttt{x3}},\overline{\texttt{x4}}\} \leftarrow \texttt{v2};\texttt{C}]$.

E. Annotation Phase

The annotation phase annotates a query by indicating which sub-queries have to be sent to which data sources. This phase does not change the query structure but add it additional information. We use a tree structure storing an IQL query when it is implemented by the AutoMed API. The root node of a query tree can be used to contain the annotation information.

F. Localization phase

The localization phase takes each sub query, transforms it into a local query expression, for example in relational algebra, and sends the query to its data source to evaluate. If there is a common expression need to be computed repeatedly, the redundant computing can be eliminated by storing the computing result.

V. Conclusion

This paper discusses an comprehension-based approach to query optimization, mainly focusing on SPJ queries without considering communication costs. We use a comprehension-based intermediate query language, the AutoMed IQL, representing high-level query languages, and apply query optimization rules on IQL queries. Our approach is expected to be extended in following three scenarios in future work.

(1) *Query optimization beyond SPJ queries* As described in Section III, the AutoMed IQL has expressibility of expressing aggregation and group functions. Reference [13] investigates formal foundations for optimizing aggregation and group functions in DBPL, which can be involved to extend our approach to handle functions beyond SPJ.

(2) *Query optimization beyond relational systems* AutoMed adopts a low-level data model which can be used to define primary high-level data models, such as relational, ER, OO, XML and flat-file, and the AutoMed IQL also has the capability representing query languages like SQL and OQL. Thus, the input queries of the comprehension phase and the output queries of the localization phase described above could be the ones beyond relational systems. In addition, by adding different query translation wrappers into the comprehension phase and the localization phase, multiple query languages can be handles in our system simultaneously, so that our approach can be extended into heterogeneous database environments.

(3) *Query optimization beyond centralized database environments* As described above, we have the gathering and annotation phases to indicate which sub-queries need to be sent to the corresponding data sources. This functionalities can be used to extend our approach into parallel database environments by considering communication costs among data sources. In addition, [11] also discusses how AutoMed can be applied in peer-to-peer data integration settings. Thus, with the extension in this paper, our query optimization approach is readily applicable in peer-to-peer and semi-structured data integration environments.

Acknowledgements

This work is supported by 973 Program (2006CB701305).

References

[1] J. Albert. Algebraic properties of bag data types. In *Proc. VLDB'91*, pages 211–219. Morgan Kaufmann, 1991.
[2] J. Albrecht, W. Hümmer, etc. Query optimization by using derivability in a data warehouse environment. In *Proc. DOLAP'00*, p. 49–56, 2000.
[3] P. Buneman, L. Libkin, D. Suciu, V. Tanen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
[4] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proc. the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43. ACM Press, 1998.
[5] H. Fan. *Investigating a Heterogeneous Data Integration Approach for Data Warehouses*. PhD thesis, Birkbeck College, Univ. of London, 2005.
[6] T. Grust and M.H. Scholl. How to comprehend queries functionally. *JIIS*, 12(2-3):191–218, 1999.
[7] R. Guravannavar, H. S. Ramanujam, etc. Optimizing nested queries with parameter sort orders. In *Proc. VLDB*, p. 481–492, 2005.
[8] Waqar Hasan, Daniela Florescu, and Patrick Valduriez. Open issues in parallel query optimization. *SIGMOD Record*, 25(3):28–33, 1996.
[9] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.
[10] Guy M. Lohman, C. Mohan, etc. Query processing in R*. In *Query Processing in Database Systems*, pages 31–47. Springer, 1985.
[11] P. McBrien and A. Poulovassilis. Defining peer-to-peer data integration using both as view rules. In *Proc. DBISP2P*, LNCS 2944, 91–107, 2003.
[12] J. McHugh and J. Widom. Query optimization for XML. In *Proc. VLDB'99*, pages 315–326, 1999.
[13] A. Poulovassilis and C. Small. Formal foundations for optimising aggregation functions in database programming languages. In *Proc. DBPL'97, LNCS 1369*, p. 299–318, 1997.
[14] P. Selinger, M. Astrahan, etc. Access path selection in a relational database management system. In *Proc. ACM SIGMOD*, p. 23–34, 1979.
[15] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proc. VLDB*, pages 355–366, 2006.
[16] P. W. Trinder. Comprehensions, a query notation for DBPLs. In *Proc. the 3rd Int'l Workshop in DBPLs*, pp 55–68. Morgan Kaufmann, 1991.