Advance Access publication on January 29, 2008 doi:10.1093/comjnl/bxm105

The Collective Index: A Technique for Efficient Processing of Progressive Queries

QIANG ZHU[1,*], BRAHIM MEDJAHED[1], ANSHUMAN SHARMA[1] AND HENRY HUANG[2]

[1]*Department of Computer and Information Science The University of Michigan, Dearborn, MI 48128, USA*
[2]*Research and Advanced Engineering Ford Motor Company, Dearborn, MI 48121, USA*
**Corresponding author: qzhu@umich.edu*

The emergence of modern data-intensive applications requires sophisticated database techniques for processing advanced types of user queries on massive data. In this paper, we study such a new type of query, called progressive queries. A progressive query is defined as a set of interrelated and incrementally formulated step-queries. A step-query in a progressive query PQ is specified on the fly based on the results of previously-executed step-queries in PQ. Hence, a progressive query cannot be formulated in advance before its execution, which raises challenges for its processing and optimization. We introduce a query model to characterize different types of progressive queries. We then present a new index structure, called the collective index, to efficiently process progressive queries. The collective index technique incrementally evaluates step-queries via dynamically maintained member indexes. Utilizing the special structure of a collective index, the (member) indexes on the input relation(s) of a step-query are efficiently transformed into indexes on the result relation. Algorithms to efficiently process single-input (unary) linear and multiple-input (join) linear progressive queries based on the collective index are presented. Our experiment results show that the proposed collective index technique outperforms the conventional query processing methods in processing progressive queries.

Keywords: progressive query; query processing and optimization; index structure; index maintenance; algorithm; performance

Received 12 April 2007; revised 12 November 2007

1. INTRODUCTION

The recent years have witnessed the emergence of advanced applications (e.g. biology, astronomy, and telematics) that produce an avalanche of data consisting of billions of records [1, 2]. Scientific instruments and computer simulations are creating very large data sets that are approximately doubling each year. These data-intensive applications require sophisticated database techniques for processing advanced types of queries. In many such applications, users routinely perform data analysis (queries) in steps by using the results of previous steps in subsequent analysis (queries) [2, 3]. The drug discovery process is an example of complex scientific problem that involves querying vast amounts of data (e.g. structural and functional information on genes, sequence searching and retrieval, toxicity, efficacy and viability) in an incremental fashion [4]. This process is characterized by a well-defined set of phases (e.g. target identification, target validation); each phase includes the querying of data produced by the previous phases. Another example is protein identification in bioinformatics. Assume that a biologist obtains a novel DNA sequence nothing is known about. He/she first runs an alignment to identify that sequence. The alignment tool executes a query that returns a vast amount of significantly similar sequences. The scientist would like then to validate the returned data as being retrieved from reliable sources whose databases are subject to routine curation. Subsequent steps may include comparing his/her experiment results with those obtained by other scientists on the returned data. Geo-scientists have generated massive volumes of earth science data for decades [5]. This data is usually accessed via a large number of complex multi-step queries. Consider the following example query: 'What are the distribution,

U/Pb zircon ages and the gravity expression of A-type plutons in Virginia? Are these A-type igneous rocks in Virginia related to a hot spot trace? Are the bodies tabular in 3-D geometry?' A number of steps are involved in answering the above mentioned query such as identifying A-type bodies and accessing their published ages [5]. The need for incrementally querying large volumes of data spans other application domains such as e-commerce (e.g. customers searching products in a vast electronic catalog), decision support systems (e.g. managers accessing and analyzing large corporate data) and multimedia applications (e.g. users browsing songs in a world-wide music database).

Unlike in traditional applications, queries in the aforementioned applications are formulated in more than one step (progressively). We call such a query a progressive query, and each of its steps a step-query. As an illustration, assume that a user wanted to select a song to play from a world-wide music database including millions of songs and lyrics. He/she first queried the database to list all songs released in the past 10 years. He/she found that there were too many such songs in the database. He/she then requested to narrow down the list by imposing a further condition on the genre. This example showcases two important characteristics of progressive queries. First, a progressive query cannot be known beforehand. Users specify step-queries on the fly based on the results of previously-executed step-queries. Second, step-queries are often executed over large data sets. The results of a step-query may not be held in memory, which requires access to disk for executing such a step-query. These characteristics raise new challenges for efficiently processing such queries. For example, no index on the result (relation) of a step-query is available for processing the next step-query. It is clear that dynamically creating indexes from scratch at each step would incur unacceptable overhead.

To tackle the above challenge, we present a novel collective index technique for efficiently processing progressive queries. The key idea is to employ a special index structure that allows a collection of member indexes (B^+-trees [6, 7]) on an input relation of a step-query to be efficiently transformed into (member) indexes on the result relation, which can then be utilized by the following step-query. We also introduce a query model to characterize different types of progressive queries. The main reason for choosing the B^+-tree as the member index structure in our technique is that it is the most popular index technique employed in current database management systems (DBMSs). Although the hashing (index) technique is also extensively adopted in DBMSs, each relation can have only one hashing structure based on one attribute. Transforming such a hash structure from an input relation to the result relation based on the same attribute may not help improve the processing efficiency for a progressive query since the progressive query may retrieve data from a relation based on different attributes throughout its steps. Although applying indexes to speed-up query processing has been well studied, using a special structure to efficiently maintain member indexes for processing special progressive queries is our new idea. To our knowledge, no similar work has been reported in the literature.

The work that is most related to ours in the literature includes query answering using views [8–11], query processing for continuous queries [12–15, 25], and adaptive (dynamic) query processing/optimization [16–20]. The problem of query answering using views is to find efficient methods of answering a query using a set of previously defined materialized views over the database, rather than directly accessing the database relations [10]. One important difference with progressive queries is that views are materialized *a priori* while the results returned by step-queries of a progressive query are obtained dynamically during the progressive query execution. On the other hand, a step-query cannot be formulated without knowledge of the result(s) of its previously-executed step-query(ies). This makes the major methods for executing queries over views based on rewriting techniques irrelevent. Continuous queries require the repeated execution of a query over a continuous stream of data [13]. The main difference with progressive queries is that a continuous query is formulated at once (although data is dynamic) while a progressive query is formulated in several steps. The idea in adaptive query optimization is to exploit information that becomes available at query runtime and adapt the query plan to changing environments during execution. While the adaptive query optimization problem may be seen as 'progressive' (performed at compile-time and run-time), queries are, however, formulated at once ('non-progressive').

The rest of the paper is organized as follows. Section 2 introduces a query model to characterize different types of progressive queries. Sections 3 and 4 present a collective index technique to efficiently process two types of progressive queries. We discuss the technique for processing single-input (unary) progressive queries first, and then extend it to handle multiple-input (join) progressive queries. Section 5 reports experiment results that demonstrate the efficiency of the collective index technique. The last section summarizes the conclusions and future research directions.

2. THE PROGRESSIVE QUERY MODEL

A progressive query (PQ) is formulated in several steps, each step is called a step-query (SQ). Users submit the first step-query SQ_1 on one or more so-called external relations. Based on the result (relation) of SQ_1, they submit a second step-query SQ_2 using as input the result returned by SQ_1's execution. SQ_2 may use additional external relations as input. Likewise, the third step-query SQ_3 is formulated based on the result of SQ_2's execution. SQ_3 may also use the result of SQ_1's execution and external relations as input.

This process is repeated recursively until submission of the last step-query. Users decide about the next step to perform based on their domain expertise. They may also stop this process at anytime if they consider the results returned so far as 'satisfactory'. However, neither the query processor nor the users know *a priori* the step-queries to be submitted. Users rather formulate their step-queries on the fly based on the results returned by the previous step-queries and their expertise in interpreting those results.

EXAMPLE 1. Let us consider a progressive query PQ composed of three step-queries defined as follows:

- SQ_1: $R_1' = \sigma_{\text{cond } 1}(R_1)$.
- SQ_2: $R_2' = \pi_{x,\, y}\,(\sigma_{\text{cond } 2}\,(R_1' \bowtie_{\text{cond } 3} R_2))$.
- SQ_3: $R_3' = \pi_{x,\, z}\,(R_2' \bowtie_{\text{cond } 4} R_3)$

The user first submits SQ_1. He/she then submits SQ_2 based on SQ_1's result R_1'. Finally, he/she submits SQ_3 based on SQ_2's result R_2'. R_1, R_2 and R_3 are external relations.

2.1. Dependencies between step-queries

A progressive query PQ consists of a set of inter-related step-queries $\{SQ_1, SQ_2, \ldots, SQ_n\}$. Each step-query is executed over a set of relations and returns one single relation as a result. Result(SQ_i) and Domain(SQ_i) ($i = 1,\ldots n$) refer to the relation returned by SQ_i's execution and the set of relations on which SQ_i is executed, respectively. Domain(SQ_i) is the union of two domains E-Domain(SQ_i) and I-Domain(SQ_i), i.e. Domain (SQ_i) = E-Domain(SQ_i) $\cup$ I-Domain(SQ_i). E-Domain(SQ_i), external domain of SQ_i, contains the set of external relations used by SQ_i, i.e. relations not returned by any previously executed step-query. I-Domain(SQ_i), internal domain of SQ_i, contains relations returned by the execution of other PQ's step-queries SQ_j ($j \neq i$) executed before SQ_i.

The step-queries of a given PQ are related by dependencies. SQ_i depends on SQ_j if SQ_i uses as input the result of SQ_j's execution, i.e. $\{\text{Result}(SQ_j)\} \subseteq$ I-Domain(SQ_i). We model PQ by a dependency graph DG(PQ) = (V, E) by using dependencies among step-queries. DG(PQ) is a directed acyclic graph (i.e. the step-queries are not recursive). V is the set of step-queries $\{SQ_1, SQ_2, \ldots, SQ_n\}$ that define PQ. We create an edge $SQ_j \rightarrow SQ_i$ in E iff SQ_i depends on SQ_j. We say that SQ_j is a source of SQ_i. DG(PQ) contains some step-queries that do not have sources. We call such nodes as the initial step-queries and refer to them by the set Initial(PQ). Formally, $SQ_i \in$ Initial(PQ) iff SQ_i has no incident to it in DG(PQ). The internal domain of an initial step-query SQ_i is empty: I-Domain(SQ_i) = $\emptyset$. However, its external domain contains at least one relation: E-Domain(SQ_i) $\neq \emptyset$. The execution of a PQ returns one single relation as a final result. Hence, DG(PQ) has one single sink vertex called the final step-query FSQ(PQ). Finally, the relation returned by each step-query of PQ (except FSQ(PQ)) is used as input by at least another PQ's step-query. This means that DG(PQ) is strongly connected; for each vertex SQ_i of DG(PQ), there is a path from SQ_i to FSQ(PQ).

EXAMPLE 2. Let us consider the PQ given in Example 1. This PQ contains two dependencies $SQ_1 \rightarrow SQ_2$ and $SQ_2 \rightarrow SQ_3$ so that:

- Initial(PQ) = $\{SQ_1\}$ and FSQ(PQ) = SQ_3.
- I-Domain(SQ_1) = $\emptyset$; I-Domain(SQ_2) = $\{R_1'\}$; I-Domain(SQ_3) = $\{R_2'\}$.
- E-Domain(SQ_1) = $\{R_1\}$; E-Domain(SQ_2) = $\{R_2\}$; E-Domain(SQ_3) = $\{R_3\}$.

2.2. Types of progressive queries

We identify three types of progressive queries based on the structure of their DGs and number of inputs per step-query. The structure of a DG may be linear (Types 1 and 2) or non-linear (Type 3). A PQ may be single-input (Type 1) or multiple-input (Types 2 and 3). Fig. 1 depicts the three types of typical progressive queries, where solid and dashed lines refer to external and internal relations, respectively.

Type 1. single-input linear (Fig. 1(a)). Single-input linear PQs are defined by two properties. First, DG(PQ) is linear; each non-initial step-query has one single source. The graph contains one single initial step-query. Indeed, the existence of two or more initial step-queries implies that at least one step-query has two sources since DG(PQ) is strongly connected. Second, each step-query SQ_i uses one single relation as input. If SQ_i is initial, then its external domain is a singleton. Otherwise, SQ_i's external domain is empty. In other words, none of the step-queries of a progressive query of Type 1 contains a join operation which requires at least two relations as input. Formally, a single-input linear PQ is defined by the following:

- $\forall\, i$: $(SQ_j \rightarrow SQ_i)$ and $(SQ_k \rightarrow SQ_i) \Rightarrow j = k$; and
- if $SQ_i \in$ Initial(PQ): |E-Domain(SQ_i)| = 1; and
- $\forall\, SQ_i \notin$ Initial(PQ): E-Domain(SQ_i) = $\emptyset$.

Type 2. multiple-input linear (Fig. 1(b)). Multiple-input linear PQs are defined by two properties. First, DG(PQ) is linear (see Type 1). Second, there is at least one step-query SQ_i that uses more than one relation as input. If SQ_i is an initial step-query, then SQ_i's external domain contains two or more relations. Otherwise, SQ_i's external domain contains at least one relation. In the latter case, SQ_i uses as input at least one relation from each of I-Domain(SQ_i) and E-Domain(SQ_i). In other words, SQ_i represents a join operation between two or more relations (see Example 1). Note that a query of this type requires at least one (not necessarily every) step-query to have one or more external relations. Formally, a multiple-input linear PQ is defined by the following:

- $\forall\, i$: $(SQ_j \rightarrow SQ_i)$ and $(SQ_k \rightarrow SQ_i) \rightarrow j = k$; and
- $\exists\, i$: at least one of the following properties is true:

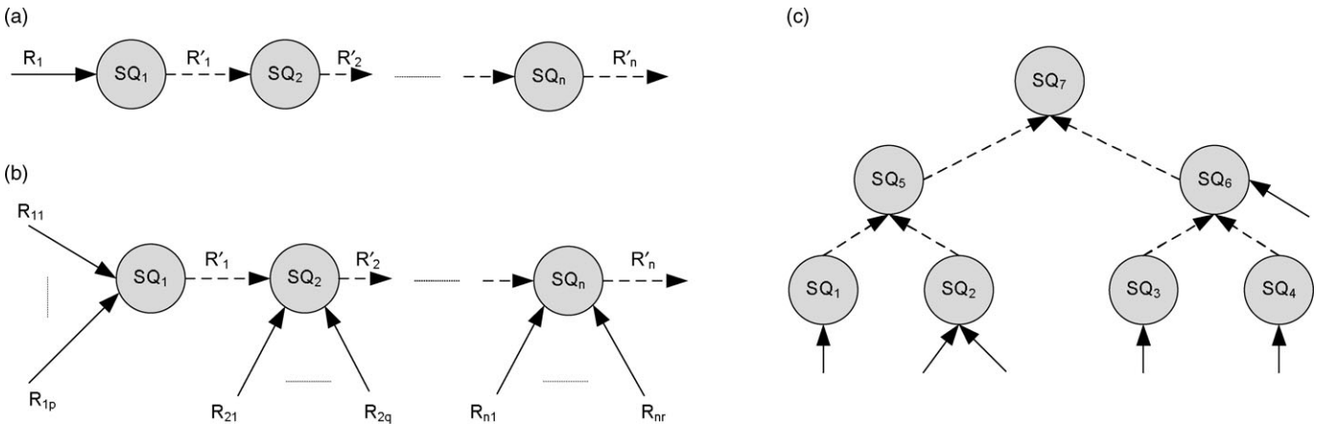

FIGURE 1. Different types of progressive queries: (**a**) Type 1: single-input linear; (**b**) Type 2: multiple-input linear; (**c**) Type 3: non-linear.

— $SQ_i \in$ Initial (PQ) and |E-Domain (SQ_i)| ≥ 2; or
— $SQ_i \notin$ Initial (PQ) and E-Domain (SQ_i) $\neq \emptyset$.

Type 3. non-linear (Fig. 1(c)). The dependency graph of a non-linear PQ is non-linear; that is, at least one step-query SQ_i in PQ has two sources SQ_j and SQ_k. Non-linear PQs are necessarily multiple-input since the internal domain of SQ_i contains at least two relations resulting from SQ_j's and SQ_k's executions, respectively. Formally, a non-linear PQ is defined by the following:

- $\exists\, i, j, k$: $(SQ_j \rightarrow SQ_i)$ and $(SQ_k \rightarrow SQ_i)$ and $(j \neq k)$.

Because of space limitation, our focus in the rest of this paper is on progressive queries of Types 1 and 2. Details about dealing with queries of Type 3 will be discussed in a future paper.

2.3. Potential strategies for processing progressive queries

We assume that a progressive query is executed on one or more large external relations. To achieve efficient query processing, we also assume that there is one or more indexes (B^+-trees) for each external relation. There are several potential strategies for processing a progressive query, including the following.

Strategy 1: Repeated evaluation via query merging (*REQM*). In this strategy, for each step i, a new query Q_i on the (original) external relations is evaluated, where Q_i is dynamically formed by merging step-query SQ_i at step i with all its previous step-queries. In other words, no result of a previous step-query is utilized to process any following step-query. For example, for the progressive query in Example 1, the query performed by this strategy at step 2 is: $\pi_{x,\,y}\,(\sigma_{\text{cond 2}}\,((\sigma_{\text{cond 1}}\,(R_1))\bowtie_{\text{cond 3}} R_2))$ rather than the given step-query SQ_2. As another example, assume that we have the following step-queries for a progressive query of Type 1:

$$SQ_i = \sigma_{F_i}(R'_{i-1}), \quad i \geq 1, \tag{1}$$

where R'_{i-1} is the result relation of step-query SQ_{i-1} for $i \geq 2$; R'_0 is the (original) external relation R for the progressive query. The following merged query on R is evaluated at each step $i \geq 1$:

$$Q_i = \sigma_{F_1 \wedge F_2 \wedge \cdots \wedge F_i}(R). \tag{2}$$

Although a merged query Q_i can utilize the index(es) on external relation(s) for its evaluation, it does not make use of any results from the previous steps. Evaluating a progressive query from scratch at each step wastes much work done in previous steps. It is clear that, when the evaluation of the progressive query has completed a number of steps and the result of the last step is relatively small comparing to the original external relation(s), reusing the result of the last step for the next step is usually much more efficient than evaluating everything from scratch. Thus this strategy, in general, does not yield satisfactory performance.

Strategy 2: Incremental evaluation via sequential scan (*IESS*). Another straightforward strategy to evaluate a progressive query is to evaluate each step-query using the result(s) of its previous step-query(ies) as input(s). Since the result of a step-query is a temporary relation, no pre-built index can be utilized to improve the performance of a subsequent step-query on this temporary relation. A sequential scan has to be used to access the temporary relation (i.e. employing the sequential scan method for a unary query and the nested-loop join method for a join query) although index(es) on external relations can still be utilized. As we know, a sequential scan is often not as efficient as an index-based access method. This strategy hardly leads to an efficient evaluation in general.

Strategy 3: Incremental evaluation via dynamically created indexes (*IEDCI*). One way to overcome the problem of the last strategy is to dynamically create a desired index(es) on the result relation of a step-query. Although a subsequent step-query (of the same progressive query) now can be evaluated efficiently using the created index(es), there is much overhead for creating dynamic indexes. Since such overhead is counted in the processing cost of the progressive query, the overall performance of query evaluation may be seriously degraded.

Strategy 4: Incremental evaluation via dynamically maintained indexes (*IEDMI*). We notice that indexes on the input relation(s) of a step-query can be efficiently transformed into indexes on the result relation of the step-query. Such efficiently maintained indexes can be used for improving the efficiency of processing a subsequent step-query. This strategy overcomes the problems of all previous strategies. In the following two sections, we will present a technique that enables this strategy by adopting a novel index structure, called the collective index. As mentioned earlier, we only consider progressive queries of Types 1 and 2 in the following discussion.

3. COLLECTIVE INDEX FOR SINGLE-INPUT (UNARY) LINEAR PROGRESSIVE QUERIES

To efficiently process progressive queries, we introduce a new index technique, called the collective index. We first present the technique (including the index structure and search algorithm) for single-input linear progressive queries (i.e. Type 1) in this section. We will discuss how to extend the technique to handle multiple-input linear progressive queries (i.e. Type 2) in Section 4.

3.1. Collective index structure

Assume that users have created, based on their application demands, a collection S of indexes (B^+-trees) on selected attributes of an external relation R. These indexes can help improve the performance of queries on relation R. Unfortunately, for a (single-input linear) progressive query PQ on R, the indexes for R are no longer useful for PQ after its first step-query. The relevant index(es) for the result relation of each step-query is needed to keep efficient query processing.

To avoid high overhead for dynamically creating indexes from scratch for the result of each step-query of PQ, we introduce a technique to efficiently transform the indexes for an input (external or previous result) relation into ones for the result relation of each step-query. The key idea is to use a novel collective index structure.

A collective index for relation R consists of a collection of member indexes on R with a special structure (see Fig. 2). Each member index is a conventional B^+-tree. However, a special router containing a set of entries is maintained. Each tuple in R has one entry in the router, and vice versa. As we know, each entry in a leaf node of a B^+-tree contains a physical tuple identifier (`Tid`), which directly gives the location/address of the corresponding tuple in R (see the relevant solid links in Fig. 2). On the other hand, given a physical `Tid`, it can also be used to obtain (indirectly) the location/address L of the corresponding entry e in the router via a mapping (see the dashed arrows in Fig. 2); i.e. there exists function φ such that $L = \varphi(\texttt{Tid})$. Specially, assume $\texttt{Tid} = (i - 1) * \texttt{tl}$ (i.e. the offset of the tuple in the R file), where i is the tuple number and `tl` is the tuple length; then $L = (\texttt{Tid/tl}) * \text{el}$ (i.e. the offset of the entry in the router file), where `el` is the entry length. If a tuple from R is put into the result relation of a step-query, its corresponding router entry contains a temporary `Tid` (`TempTid`) pointing to the tuple in the result relation. For the first step-query, the physical `Tid`s in the leaf node entries of the relevant member index are used to directly fetch tuples from the input external R. For a subsequent step-query, temporary `Tid`s in the router entries identified by the relevant physical `Tid`s are used to fetch tuples from the corresponding input temporary relation R'. In the latter case, the member indexes using temporary `Tid`s are considered to be on temporary relation R'.

This index structure facilitates an efficient transformation (maintenance) of the member indexes on an input relation into the member indexes on the result relation of a step-query, based on the fact that the leaf node entries from all member indexes that correspond to the same tuple will point (indirectly/logically) to the same entry in the router. For a given step-query, if a tuple in the input relation satisfies the query condition, it is put into the result relation. Without the collective index structure, to maintain a collection of conventional B^+-trees, we would have to go through each individual index and change the relevant leaf node entry to point (directly) to the corresponding tuple in the new result relation. With the collective index structure, we only need to make the relevant entry in the router point to the tuple in the result relation, and then all member indexes have their relevant leaf node entries automatically point (indirectly) to the right tuple in the new relation, which greatly reduces the index maintenance/transformation cost.

There are several difficulties in maintaining indexes for processing progressive queries. First of all, we cannot lose the physical `Tid`s of the tuples in the original external relation R after index transformations for step-queries. This is because the collective index should be used for many progressive queries rather than only one. To overcome this difficulty, we keep physical `Tid`s in the leaf node entries of each member index unchanged, update only temporary `Tid`s in the router entries after each step-query, and use a function to link a leaf node entry to its corresponding router entry.

Secondly, we need to efficiently identify the router entries that are valid for the current progressive query. Since a collective index may be used for multiple progressive queries, some

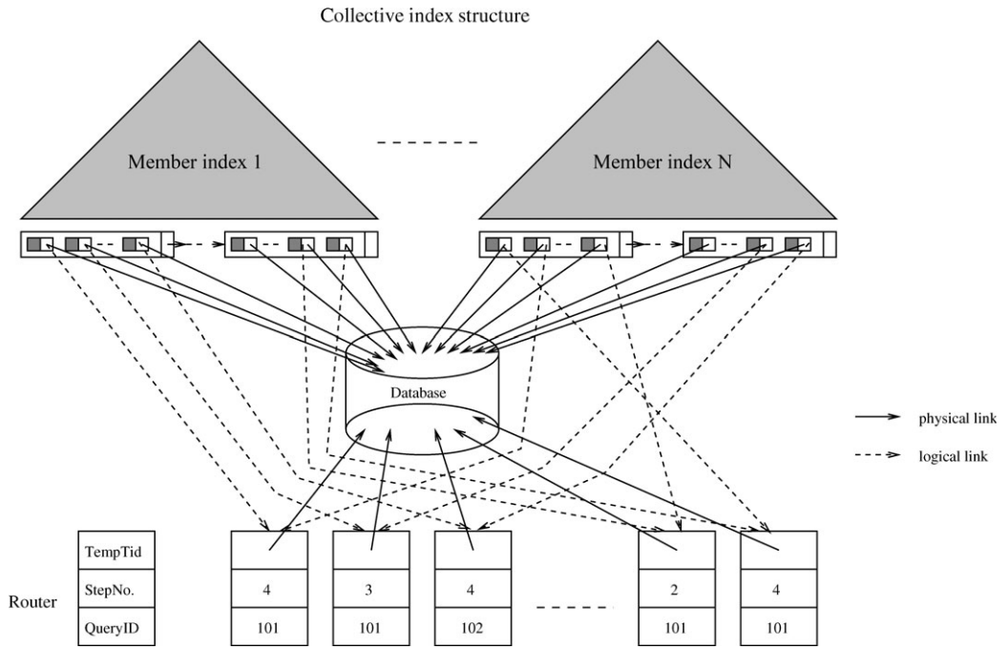

FIGURE 2. Structure of collective index.

router entries may be left with old `Tid`s for the result relations of the previous progressive queries when a new progressive query is processed. Instead of re-initializing all router entries every time when a new query is processed, which is inefficient, we add a field (i.e. `QueryID`) in each router entry to store the identifier of the current query. We assume that each progressive query has a unique query identifier. The `QueryID` field in each router entry needs to be initialized to null only when the collective index is created. No further initialization (unless all possible identifiers are exhausted) is needed when different queries are processed, since this field of a relevant router entry is properly set during the processing of a query. Specifically, during the processing of the first step-query of the given progressive query, this field is ignored since all router entries can be used. When a qualified tuple t is put into the result relation R_1' of the first step-query, the `TempTid` field of the relevant router entry is set to the `Tid` of tuple t in R_1' and, in the meanwhile, the `QueryID` field is set to the identifier of the current progressive query. Since only tuples in R_1' can be in the result relation of the next step-query, no router entry whose query identifier is different from the current one can be used in the subsequent processing.

Thirdly, we also need to efficiently identify the router entries that are valid for the current step-query of the given progressive query. As we know, some router entries may not be used for the tuples of the result relation of the step-query before the current one. However, only those router entries used for such tuples are valid for processing the current step-query. To identify the router entries valid for the current step-query, we add a field (i.e. `StepNo`) in each router entry to indicate the step-query for which this entry is used. This field is ignored when the first step-query is processed. When a qualified tuple is put into the result relation of the step-query, this field is set to the current step number. When the next step-query is processed, only router entries whose `QueryID` is current and whose `StepNo` is the step number of the last step are valid for the current processing. Since this field is set while a qualified tuple is found during the processing of a step-query, no separate initialization is needed at the beginning of each step-query processing.

From the above discussion, we can see that the main reasons why the maintenance of a collective index is efficient are: (1) the member indexes share the same set of router entries so that any change in a router entry is automatically reflected in all the member indexes and (2) a careful design eliminates necessity of many costly re-initializations for each progressive query and its step-queries.

EXAMPLE 3. Let us consider a step-query SQ_3: $\sigma_{R_{2,a}' > 10}(R_2')$ at step 3, executed over the result relation R_2' of the previous step-query at step 2. It can be processed efficiently by utilizing the collective index CI_2' on R_2' to produce a new result relation R_3'. An updated collective index CI_3' on R_3' is obtained from CI_2' during the processing. Fig. 3 shows some router entries for CI_2' and CI_3'. Assume that `QueryID` for the current progressive query is 101. E_{21}, E_{23} and E_{24} in the router of CI_2 are valid entries for SQ_3, while E_{22} is not since its `QueryID` is not 101. Valid entries E_{31} and E_{34} in the router of CI_3' are obtained from E_{21} and E_{24}, respectively. E_{33} (coming from E_{23}) becomes invalid (i.e. not satisfying the query condition). This is indicated by an old `StepNo` (i.e. 2).

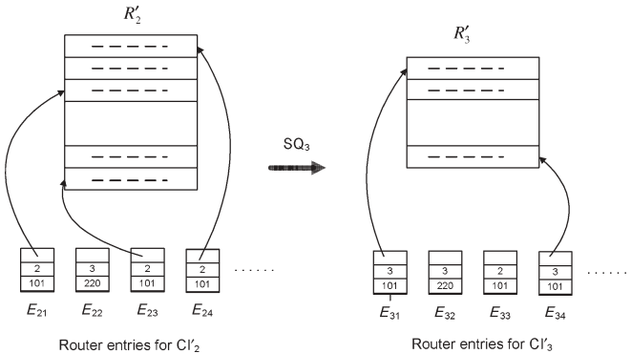

FIGURE 3. Example: processing a Type 1 query.

3.2. Query processing using collective indexes

Assume that a collective index is created for the input external relation R of a given single-input linear progressive query PQ. For simplicity, we assume that one member index is used for each step-query of PQ in the following discussion. That is, the selection condition F_i in Formula (1) is $R'_{i-1}.a = C$ or $C_1 \leq R'_{i-1}.a < C_2$, where a is an indexed (by a member index) attribute of relation R'_{i-1} (recall $R'_0 = R$); C, C_1, C_2 are the constants in the domain of a. Note that our discussion can easily be extended to allow multiple member indexes for each step-query. The searching and index-maintaining algorithm for step-query SQ_i of PQ at step i ($i \geq 1$) is given as follows:

Algorithm 1: Step query processing via collective index-Type 1 (SQPCI-1)

Input: (1) current query identifier `CurrentQid`; (2) current step number i ($\geq$1); (3) step-query SQ_i; (4) input relation R'_{i-1} for SQ_i; (5) collective index CI'_{i-1} for R'_{i-1}.

Output: (a) result relation R'_i of SQ_i; (b) transformed collective index CI_i' for R'_i.

Method:

(1) Initialize result relation R'_i to empty;
(2) Identify a member index I in CI'_{i-1} that can be used for processing SQ_i;
(3) Use the relevant key/range in the qualification condition of SQ_i to traverse I from the root to one or more leaves to find a set E of leaf node entries for candidate tuples in the result relation;
(4) **if** E is not empty **do** // there is a match
(5) **for** each leaf node entry `le` in E **do**
(6) fetch corresponding router entry `re` located by φ(`le.Tid`);
(7) **if** $i = 1$ **or** (`re.QueryID` = `CurrentQid` **and** `re.StepNo` = $i - 1$)**then** // the router entry is //valid for the current step
(8) **if** $i = 1$ **then**
// access external tuples via physical
// `Tid`s in the 1st step
(9) Fetch the tuple t located by `le.Tid` from R'_{i-1};
(10) Let `re.QueryID` = `CurrentQid`;
(11) **else**
// access tuples from the last step via temporary
// `Tid`s
(12) Fetch the tuple t located by `re.TempTid` from R'_{i-1};
(13) **end if**
(14) Put t into R'_i;
(15) Let `re.TempTid` be the new `Tid` of t in R'_i;
(16) Let `re.StepNo` = i;
(17) Update the router with modified entry `re`;
(18) **end if**
(19) **end for**
(20) Let CI_i' be CI_{i-1}' with the revised router entries;
(21) **else**
(22) Let CI_i' be an empty collective index;
(23) **end if**
(24) **return** R_i' and CI_i'.

Algorithm SQPCI-1 behaves just like a normal index access method except that (i) tuples in an input relation may be accessed indirectly via router entries (lines 6 and 12), and (ii) some index maintenance work (e.g. lines 10, 15 and 16) is piggybacked on the access process. Hence extra memory and CPU overheads are needed for the collective index technique. The extra space required by a collective index is for its router entries, which amounts to: $(L_1 + L_2 + L_3)$ * $|R_0|$, where L_1, L_2 and L_3 are the sizes of `TempTid`, `StepNo` and `QueryID` (i.e. $L_1 + L_2 + L_3$ is the entry size), respectively; and $|R_0|$ is the cardinality of the input external relation R_0 for the given progressive query. Since L_1, L_2 and L_3 are small (e.g. 4 bytes, 1 byte and 1 byte, respectively), the extra space for the router entries is typically not large. For example, a relation of 10 millions tuples requires only 60 MB for its router, which could be held entirely in memory in today's computers. In case the available memory is not enough to hold all entries in the router, some additional I/O cost is required. The buffering strategies discussed below can be adopted to reduce such I/O cost. The extra CPU overhead for the collective index technique consists of the cost for accessing and validating the relevant router entries corresponding to the qualified leaf node entries in E (lines 6 and 7) and the cost for updating/maintaining these router entries (lines 10, 15, 16 and 17). Hence the extra CPU overhead is $O(|E|)$. Like any other index techniques, the collective index is applicable to a step-query with a small selectivity, i.e. small $|E|$ (relative to $|R_0|$). Thus the extra CPU is very small, especially comparing to the I/O cost which is usually dominant during database query processing [21, 22].

To minimize extra I/O cost, we incorporate some buffering strategies into the query processing algorithm. The above algorithm SQPCI-1 requires one router disk read (at line 6) and one router disk write (at line 17) for each identified leaf node entry (from line 3) in the worst case. To reduce the number of times that the same router block is read/written, we enhance the algorithm so that it processes all leaf node entries in the same leaf node block (or a fixed number of such blocks) that have corresponding router entries in the same router block B together (instead of individually) while B is in the memory. Furthermore, we employ a bitmap to indicate the router blocks that are never accessed in the current step. In a subsequent step, if a leaf node entry `le` has an associated router entry in such a router block B', we know immediately that `le` is invalid without having to access B', leading to reduced I/O cost. We call the algorithm incorporating these strategies a buffered SQPCI-1.

Since an index access method is, in general, more efficient than the sequential scan method, our collective index access method usually has a better performance than the direct query processing Strategy IESS in Section 2.3, as we will see in Section 5.1.

4. SUPPORTING MULTIPLE-INPUT (JOIN) LINEAR PROGRESSIVE QUERIES

This section is devoted to multiple-input linear progressive queries (i.e. Type 2). As mentioned earlier, progressive queries of this type include at least one step-query SQ_i that has two or more relations as input. If $i = 1$, then all SQ_i's inputs are external relations. Otherwise, one of the SQ_i's inputs is the result relation R'_{i-1} of the previous step $(i - 1)$; the other inputs are external relations. SQ_i can hence be seen as a join query between two or more relations. For simplicity, we assume that each step query of a multiple-input linear progressive has two input relations in the remainder of this paper. Our discussion can easily be extended to more general cases.

Several algorithms have been proposed in the literature for computing the join of relations [23]. In this paper, we adopt the indexed nested-loop join (also called the index-based join method) because of its simplicity, popularity and efficiency. Assume that SQ_i involves two relations R_{out} (called outer relation) and R_{in} (called inner relation). R_{out} and R_{in} are being joined based on an equal value for two attributes $R_{out}.a$ and $R_{in}.b$.SQ_i's output is the result relation R'_i that has all attributes from R_{out} and R_{in}. The query processor performs a sequential scan on R_{out}. For each tuple t_{out} in R_{out}, it uses R_{in}'s collective index (the member index for $R_{in} \cdot b$, in particular) to retrieve a set S of R_{in}'s tuples that satisfy the join criterion $R_{out} \cdot a = R_{in} \cdot b$. For each qualified t_{in} in S, tuples t_{out} and t_{in} are concatenated and inserted into R'_i. We will use the notation (t_{out}, t_{in}) to refer to such a concatenated tuple.

4.1. The collective index revisited

Like single-input linear progressive queries, we use a collective index structure to speed-up the processing of multiple-input linear progressive queries. The collective index consists of a set of member indexes that point to a shared router. The router structure is extended to adapt to the peculiarities of multiple-input linear progressive queries. In the case of single-input linear progressive queries, each router entry includes one single `TempTid` field. If the entry is valid, then `TempTid` is a pointer to a tuple in the relation R'_i resulting from SQ_i's execution. Assume now that SQ_i is a join over relations R_{out} and R_{in} in a multiple-input linear progressive query (R_{in} is in fact the result relation R'_{i-1} of SQ_{i-1} if $i > 1$). The same tuple t_{out} (resp., t_{in}) from R_{out} (resp., R_{in}) may appear several times in R'_i since t_{out} (resp., t_{in}) may be joined to various tuples in R_{in} (resp., R_{out}). Therefore, we need to keep a list of `TempTid`s within each router entry. The linked lists within the router dynamically evolve while processing step-queries.

Let us now consider a step-query SQ_i $(i > 1)$ executed over an external outer relation R_{out} and the inner relation R'_{i-1} (the result of SQ_{i-1}). Assume that SQ_i is a join with the condition $R_{out}.a = R'_{i-1}.b$. The query processor accesses a router entry E_K corresponding to a key value K in the member index for $R'_{i-1}.b$. The tuples pointed by the list ${L^{i-1}}_K$ of `TempTid`s in E_K are used to produce the result relation R'_i, hence creating a new list of `TempTid`s L^i_K. After resuming SQ_i's processing, another tuple from R_{out} with the condition $R_{out}.a = K$ may be fetched subsequently. Hence, the aforementioned ${L^{i-1}}_K$ from R'_{i-1}'s router is used again to populate the result relation R'_i. L^i_K is then updated to keep track of the newly obtained join results. Hence, at any time, two lists of `TempTid`s need to be maintained for each router entry E_K: L^{i-1}_K from the previous step $i - 1$ and L^i_K of the current step. For that reason, we keep the start pointers of two lists within each router entry: `EListTempTid` and `OListTempTid` (see Fig. 4). If the current step is even (resp., odd), then `OListTempTid` (resp., `EListTempTid`) contains pointers to the results of the previous step and `EListTempTid` (resp., `OListTempTid`) will contain pointers to the results of the current step. The remaining fields within a router entry (i.e. `QueryID` and `StepNo`) are similar to the ones used in single-input linear progressive queries (see Fig. 2).

Assume now that a progressive query includes four step-queries SQ_1, SQ_2, SQ_3, and SQ_4 where (i) SQ_1 joins R_1 and R_2 producing R'_1, (ii) SQ_2 joins R_3 and ${R_1}'$ producing R'_2, and (iii) SQ_3 joins R_4 and R'_2 producing R'_3. The current result relation R'_3 includes all attributes in R_1, R_2, R_3 and R_4. The next step-query SQ_4 may use any of these attributes. We should then be able to trace back the member indexes and router entries of joined tuples in the original relations R_1, R_2, R_3 and R_4. These router entries will eventually be updated after executing SQ_4 to point to the newly generated

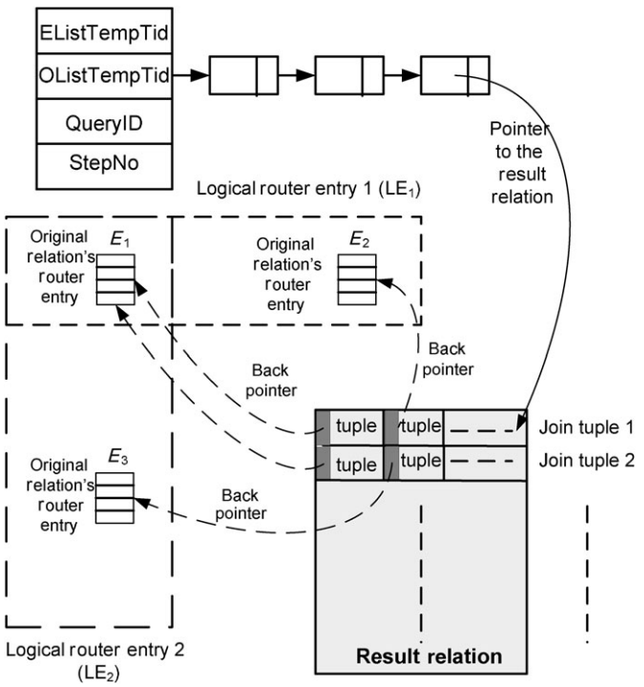

FIGURE 4. Router structure for multiple-input linear queries.

relation R'_4. For that reason, each tuple t_j in an external relation R_j is prefixed by a pointer (called a back pointer) to the corresponding entry in R_j's router (see Fig. 4). Back pointers are carried out with tuples and inserted in result relations; each time t_j is joined in SQ_i, both t_j and its back pointer are stored in the result relation R'_i.

As we know, in the case of single-input progressive queries, the collective index of a (external or result) relation R has a 'physical' router entry for each tuple in R. This is still true for the collective indexes of external relations in the case of multiple-input linear progressive queries. However, for the collective index of the result relation R'_i for a (join) step-query in a multiple-input linear progressive query, the situation is slightly different. Since a tuple t in R'_i joins the tuples from two or more original external relations, it is related to multiple physical router entries $E_1, E_2, \ldots, E_n$ from the collective indexes of the relevant external relations. One way to implement the collective index for R'_i is to create a new physical router entry for the tuple t that combines information from the original router entries $E_1, E_2, \ldots, E_n$. Clearly, this approach is inefficient. To overcome this problem, we adopt the concept of a 'logical' (or 'virtual') router entry for the collective index of R'_i. Specially, we consider that tuple t has a logical router entry LE in the collective index of R'_i, where LE consists of the original router entries $E_1, E_2, \ldots, E_n$. The router entries $E_1, E_2, \ldots, E_n$ are called the component router entries of LE. In this way, we can efficiently reuse the physical router entries from the collective indexes of the original external relations without expensive creation/materialization of new physical router entries. As a result, the collective index for R'_i is the union of the collective indexes of the original external relations with updated physical (component) router entries.

For example, let us consider the first step-query SQ_1 in a progressive query that uses R_1 and R_2 as external relations. The collective index CI'_1 of the result relation for SQ_1 contains the updated router entries in the collective indexes of R_1 and R_2. Note that the same physical router entry may belong to several logical router entries. For instance, Fig. 4 depicts two logical router entries LE_1 and LE_2. LE_1 includes two (physical) router entries E_1 and E_2 that refer to two tuples t_{out} and t_{in1} from the outer and inner relations, respectively. LE_2 includes two router entries E_1 and E_3 that refer to two tuples t_{out} and t_{in2} from the outer and inner relations, respectively. This situation corresponds to the case where the same external tuple (i.e. t_{out}) is joined with two internal tuples (i.e. t_{in1} and t_{in2}). Hence, two joined tuples (t_{out}, t_{in1}) and (t_{out}, t_{in2}) are inserted in the result relation.

4.2. Processing multiple-input linear progressive queries

We give below the algorithm for processing a step-query SQ_i in the case of multiple-input linear progressive queries. SQ_i has two input relations R_{out} and R_{in}. R_{out} is an external relation. R_{in} is an external relation for $i = 1$, otherwise (i.e. $i > 1$) it is the result relation R'_{i-1} of SQ_{i-1}. For simplicity, we assume that one member index of CI_{in} is used in SQ_i. The algorithm can be easily extended to handle join conditions that refer to several (member) indexes in R_{in}'s collective index.

Algorithm: Step query processing via collective index - Type 2 (SQPCI-2)

Input: (1) current query identifier `CurrentQid`; (2) current step number i (≥ 1); (3) step-query SQ_i; (4) outer relation R_{out} for SQ_i; (5) collective index CI_{out} for R_{out}; (6) inner relation R_{in} for SQ_i; (7) collective index CI_{in} for R_{in}.

Output: (a) result relation R'_i; (b) transformed collective index CI'_i for R'_i.

Method:

(1) Initialize result relation R'_i to empty;
(2) Identify a member index I in CI_{in} that can be used for processing SQ_i;
(3) **for** each tuple t_{out} in R_{out} //outer loop of the join
(4) Use the relevant key in the join condition of SQ_i to traverse I from the root to one or more leaves to find a set E of leaf node entries for candidate tuples in the result relation;
(5) **if** E is not empty **then** // there is a match
(6) **for** each leaf node entry `le` in E **do**
// inner loop of the join
(7) fetch corresponding router entry re_{in} located by φ(`le.Tid`);

(8) **if** $i = 1 \vee (\mathrm{re_{in}}.\texttt{QueryID} = \texttt{CurrentQid} \wedge \mathrm{re_{in}}.\texttt{StepNo} = i - 1)$ **then**
// the router entry is valid for the current step
(9) **if** even(i) **then** //identify input-output lists
// based on step number
(10) InputList = OListTempTid;
(11) OutputList = EListTempTid;
(12) **else**
(13) InputList = EListTempTid;
(14) OutputList = OListTempTid;
(15) **end if**
(16) **if** $i = 1$ **then**
// access external tuples via physical Tids in
// the 1st step
(17) Fetch the tuple t_{in} located by le.Tid from R_{in};
(18) Put (t_{out}, t_{in}) into R'_i;
// insert joined tuples in the result relation
// maintain the router entry of $\mathrm{CI_{in}}$
(19) Insert in $\mathrm{re_{in}}$.OutputList the new Tid of R_i''s (t_{out}, t_{in});
(20) Let $\mathrm{re_{in}}$.StepNo = i;
(21) Let $\mathrm{re_{in}}$.QueryID = CurrentQid ;
(22) Update the router of $\mathrm{CI_{in}}$ with modified entry $\mathrm{re_{in}}$;
// maintain the router entry of $\mathrm{CI_{out}}$
(23) Fetch the router entry $\mathrm{re_{out}}$ pointed by t_{out}'s back pointer;
(24) Insert in $\mathrm{re_{out}}$.OutputList the new Tid of R_i''s (t_{out}, t_{in});
(25) Let $\mathrm{re_{out}}$.StepNo = i;
(26) Let $\mathrm{re_{out}}$.QueryID = CurrentQid;
(27) Update the router of $\mathrm{CI_{out}}$ with modified entry $\mathrm{re_{out}}$;
(28) **else**
// access tuples from the last step via Temp-
// Tids in InputList
(29) **for** each TempTid in $\mathrm{re_{in}}$.InputList **do**
(30) Fetch the tuple t_{in} from R_{in} located by TempTid;
(31) Put (t_{out}, t_{in}) into R_i'; // insert joined
// tuples in the result relation
// maintain the router entries of CIs
// for pre-used external relations
(32) **for** each back pointer BP in (t_{out}, t_{in}) **do**
(33) Insert in BP.re.OutputList the new Tid of R_i''s (t_{out}, t_{in});
(34) Let BP.re.StepNo = i;
(35) Update the router with modified entry re;
(36) **end for**
(37) **end for**
(38) **end if**
(39) **end if**
(40) **end for**
(41) **end if**
(42) **end for**
(43) **if** R'_i is not empty **then**
(44) Let $\mathrm{CI}_{/i}$ be the union of CIs for R_{out} and R_{in} with the revised router entries;
(45) **else**
(46) Let CI'_i be an empty collective index;
(47) **end if**
(48) **return** R'_i and CI'_i.

Algorithm SQPCI-2 extends SQPCI-1 (for single-input linear progressive queries) in two ways. First, it includes two nested loops to access the outer and inner relations (lines 3 and 6, respectively). As mentioned earlier, the outer relation is scanned sequentially, while the inner relation is accessed via the collective index $\mathrm{CI_{in}}$. For the first step number, SQPCI-2 fetches the matched tuples in R_{in} via physical Tids included in the corresponding leaf nodes (lines 16 and 17). For the remaining step numbers, SQPCI-2 fetches the matched tuples in R_{in} via temporary Tids stored in the input lists included in the corresponding router entries (lines 29 and 30). The way we determine the input and output lists of each router entry depends on the step number (lines 9–15). Each pair of joined tuples (t_{out}, t_{in}) is inserted as a result tuple into R'_i (lines 18 and 31). The second extension to SQPCI-1 is related to maintaining router entries. A result tuple (t_{out}, t_{in}) inserted in R'_i may be a join of several 'initial' tuples that belong to previously used external relations. The (physical) router entry of each 'initial' tuple is accessed via the back pointer prefixing that tuple in (t_{out}, t_{in}). Each of such router entries is maintained as follows: besides updating the step number and Query ID (lines 20, 21, 25, 26 and 34), SQPCI-2 inserts the (temporary) Tid of R_i''s (t_{out}, t_{in}) into the output list of the router entry (lines 19, 24 and 33). In this way, the relevant physical router entries in the collective indexes of all previously used external relations point to the corresponding tuples in R'_i. Like algorithm SQPCI-1, in the implementation, we incorporate the similar buffering strategies (including the bitmap) to minimize the I/O cost.

EXAMPLE 4. Let us consider the first step-query $\mathrm{SQ_1}$ (of a progressive query) executed over R_{out} and R_{in} with a join condition $R_{\mathrm{out}}.a = \mathrm{R_{in}}.b$ as shown in Fig. 5. Since R_{out} and R_{in} are external relations, the lists (even and odd) of all R_{out}'s router entries and R_{in}'s router entries are empty before $\mathrm{SQ_1}$'s execution. The back pointers of tuples t_{out1}, t_{in1}, t_{in2}, and t_{in3} point to their router entries E_{out1}, E_{in1}, E_{in2}, and E_{in3}, respectively. As mentioned in algorithm SQPCI-2, R_{out} is accessed sequentially. Let us show the way SQPCI-2 is executed when tuple t_{out1} is fetched from R_{out}. Since $a = 5$ in t_{out1}, the query processor traverses R_{in}'s member index corresponding to the attribute b to look for the key value 5. Assume that three tuples t_{in1}, t_{in2}, and t_{in3} satisfy the condition $b = 5$. These

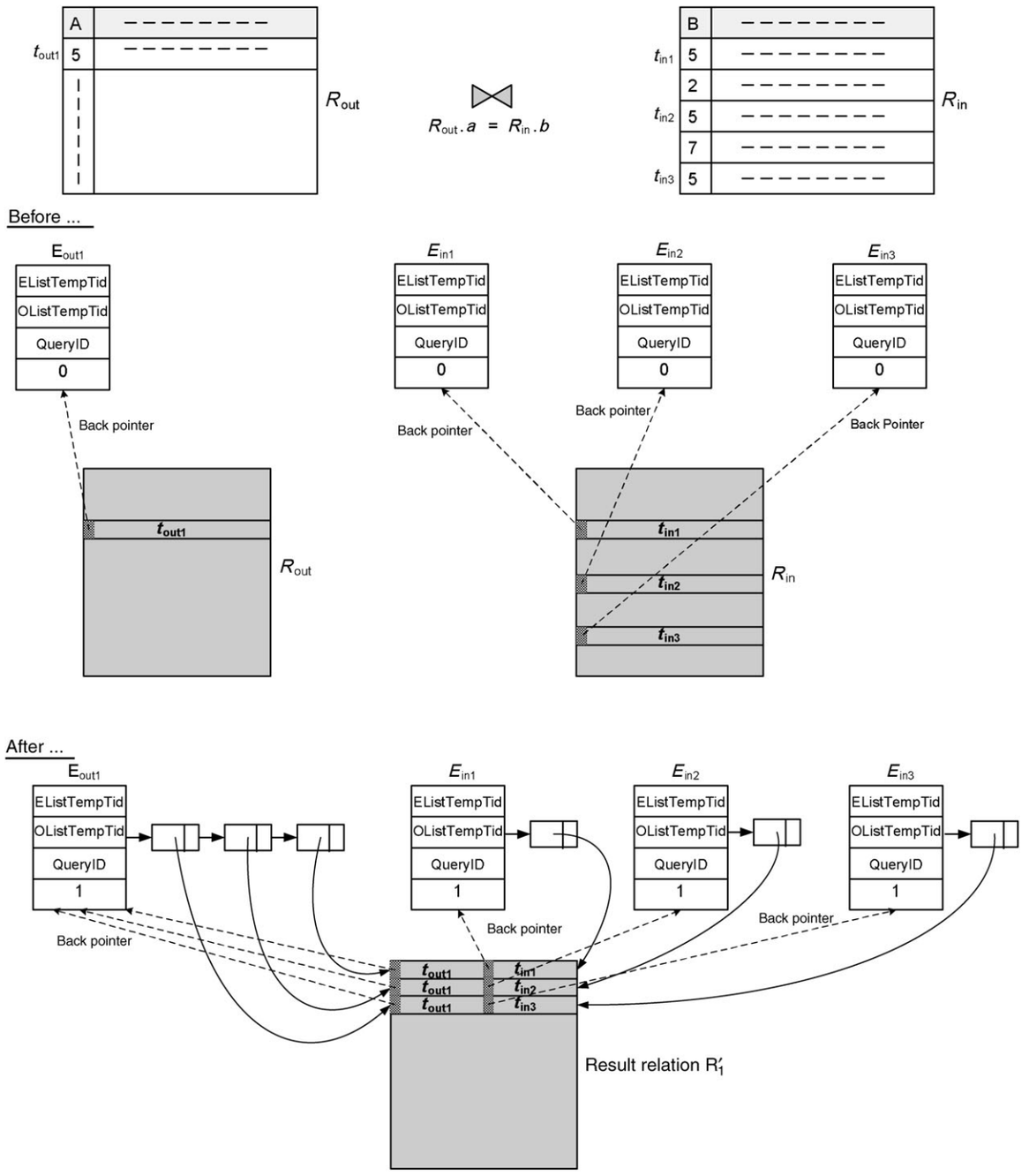

FIGURE 5. Example-processing a Type 2 query.

three tuples are retrieved via the member index. The three joined tuples (t_{out1}, t_{in1}), (t_{out1}, t_{in2}) and (t_{out1}, t_{in3}) are stored in the result relation R'_1. To keep track of the new results, the `OListTempTid` lists (since the step number is odd) of the router entries E_{out1}, E_{in1}, E_{in2} and E_{in3} are updated. The addresses of (t_{out1}, t_{in1}), (t_{out1}, t_{in2}) and (t_{out1}, t_{in3}) are inserted in E_{out1}'s `OListTempTid` (since t_{out1} is part of each result tuple). E_{in1}'s, E_{in2}'s and E_{in3}'s are updated with the address of (t_{out1}, t_{in1}), (t_{out1}, t_{in2}) and (t_{out1}, t_{in3}), respectively. The step number of each router entry is also updated (with the value 1). The maintained collective index for result table R'_1 is the union of the collective indexes of R_{out} and R_{in} with the updated component router entries such as E_{out1}, E_{in1}, E_{in2}, and E_{in3}. The logical router entry for result tuple (t_{out1}, t_{in1}), for example, consists of component router entries E_{out1} and E_{in1}. The updated information in these router entries can

be used to efficiently retrieve qualified tuples for the next step-query.

Note that the algorithm for processing single-input linear progressive queries (SQPCI-1) can easily be modified to adapt to the revised collective index structure as follows. Depending on the current step number *i*, either `EListTempTid` list or `OListTempTid` list is used to store the `TempTid` of the tuple in the result relation. However, each `TempTid` list in a router entry contains no more than one `TempTid` since each tuple in the input relation occurs at most once in the result relation (which is not the case for progressive queries of Type 2).

It is also worth pointing out that the collective index structure can be further extended to handle non-linear progressive queries (i.e. Type 3). The aforementioned idea of 'logical' router entries needs to be revised to deal with the situation where multiple result relations from several previous steps are used as input relations for the current step. On the other hand, since the execution order of the step-queries is non-linear, optimization via re-ordering and/or parallelism needs to be taken into consideration. How to efficiently processing progressive queries of Type 3 is one of our future research topics.

5. EXPERIMENTS

To evaluate the performance of the collective index technique for processing progressive queries, we conducted extensive experiments. The experiment programs were implemented in Java on a PC with Pentium 4 (3.0 GHz) CPU and 1 GB memory running Windows XP operating system. A synthetic database similar to that in [24] was used for the experiments. Specifically, values for the *i*th attribute ($1 \leq i \leq 256$) of a relation are randomly-generated integers from domain [1, 10 * i]. In other words, a lower attribute has a less number of distinct values than a higher attribute. Hence, different selectivities can be provided for queries on different attributes in a relation. We set the I/O block size to be 1 K bytes (i.e. each tuple in a relation occupies one disk block). Clearly, the collective index technique is beneficial only if its maintained collective index is utilized during progressive query processing. Thus, in the experiments, we assume that the collective index technique can utilize its maintained indexes to process the step queries in progressive queries. We evaluate the performance of the collective index technique for progressive queries of Type 1 in Section 5.1 and progressive queries of Type 2 in Section 5.2.

5.1. Performance for single-input linear progressive queries

To examine the effectiveness of the buffering strategies described in Section 3.2, we conducted experiments to compare the performance of algorithm SQPCI-1 with that of the buffered SQPCI-1. Figure. 6 shows the I/O improvement (in percentage) achieved by the buffered SQPCI-1 over the unbuffered version for step-queries with different selectivities on an input relation with 100000 tuples. There are two conflicting factors that affect the query performance. On one hand, the buffering strategies can achieve higher efficiency for a larger selectivity since buffering is more effective when more data are retrieved by a query. On the other hand, indexing is less beneficial when the selectivity becomes larger. As shown in Fig. 6, the first factor dominates the performance improvement at the beginning, while the second factor plays a more important role when the selectivity becomes larger. In the remainder of this section, we only consider the collective index algorithms with the buffering strategies.

Since Strategies REQM (i.e. applying repeated query merging) and IEDCI (i.e. dynamically creating indexes) in Section 2.3 are infeasible in practice, we compared the performance of Strategy IEDMI (i.e. using our collective index technique) with that of Strategy IESS (i.e. using the sequential scan). Since this subsection considers single-input linear progressive queries, the collective index technique employs algorithm SQPCI-1. On the other hand, two versions of IESS are considered. The first version, IESS-S1, employs the sequential scan to process all step-queries in a (single-input linear) progressive query, while the second version, IESS-S2, uses an index method to process the first step-query on the input external relation and then applies the sequential scan for processing all subsequent step-queries in the progressive query. To be fair, we assume that IESS-S2 employs a conventional B^+-tree for the first step-query without maintaining the router entries, while our collective index technique has to maintain the router entries for all step-queries including the first one. It is known that an index is beneficial for a unary

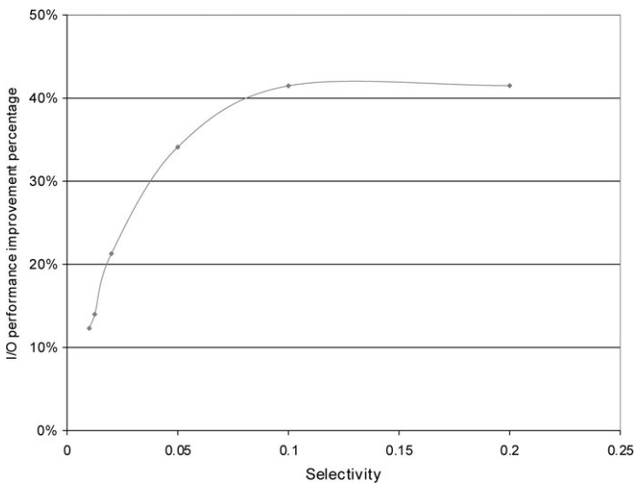

FIGURE 6. Performance effect of buffering strategies.

(step) query only if the query selectivity is small and the underlying relation is large. If this condition is not met, the collective index cannot improve query performance, just like any other index technique. Hence the selectivities of step-queries in the progressive queries in our experiments were assumed to be between 3–10%. Fig.'s 7 and 8 show the performance comparisons of the collective index technique with IESS-S1 and IESS-S2, respectively. The progressive queries used in the experiments had randomly-accessed attributes and randomly-chosen selectivities (within allowed ranges) for their step-queries on an input external relation of 10 million tuples. From the figures, we have the following observations:

- Our collective index technique always outperforms IESS-S1 in terms of the number of I/Os accessed after each step for a single-input linear progressive query. However, the improvement is reduced as more steps being processed. The reason for this improvement decreasing is because the input relation size becomes increasingly smaller for later steps of a single-input linear progressive query, which reduces the degree of improvement that an index can achieve.
- Our collective index technique outperforms IESS-S2 (on average) except for the first step of the tested queries. Since IESS-S2 uses a conventional B^+-tree without overhead for maintaining router entries, its first step is more efficient. However, our collective index technique becomes superior after the second step on average. Similar to the previous case, when a step number is large, the relevant input relation size for the step is usually small, which makes an index less beneficial. We also notice that our collective index technique does not always win for all progressive queries after the second step since the step queries for some progressive queries reduce a given input relation too quickly in their earlier steps.

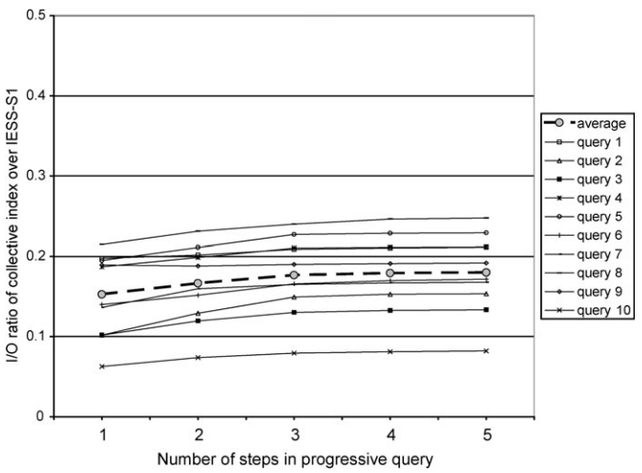

FIGURE 7. Performance comparison between collective index and IESS-S1.

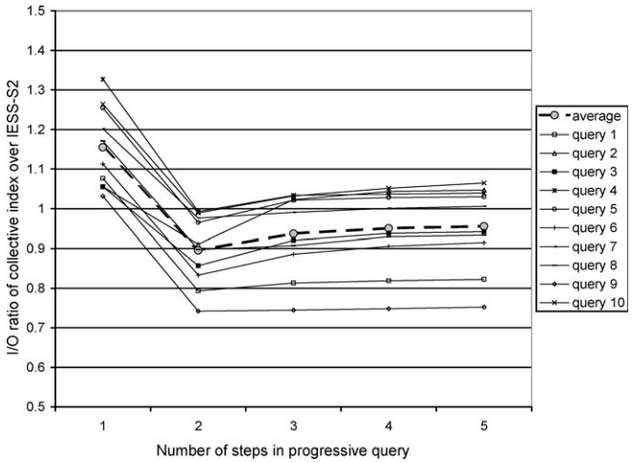

FIGURE 8. Performance comparison between collective index and IESS-S2.

5.2. Performance for multiple-input linear progressive queries

Unlike a progressive query of Type 1, a multiple-input linear progressive query that we consider here always has an external input relation besides the result relation from a previous step-query at each step. Therefore the size of the result relation of a step-query does not monotonically decrease as query processing proceeds. This increases the chance of performance improvement from indexes. However, to avoid that a join step-query produces a too small (e.g. only few matches) or too large (e.g. almost the Cartesian product) result relation in the experiments, joining attribute pairs for step-queries are randomly chosen among those that produce a result relation whose size is within the range of ten times smaller or larger than the average size of two input relations of each step-query.

Similar to single-input linear progressive queries, we compare our collective index technique with two versions of Strategy IESS: IESS-M1 and IESS-M2. IESS-M1 always employs the nested-loop join method to process each step-query in a multiple-input linear progressive query, while IESS-M2 employs the indexed nested-loop join method using the conventional B^+-tree for the first step-query and adopts the nested-loop join method for the subsequent step-queries.

Fig.'s 9 and 10 show the performance comparisons of our collective index technique with IESS-M1 and IESS-M2, respectively. Each multiple-input linear progressive query in the experiments has seven step-queries (i.e. involving 8 external relations). Since such a progressive query consists of a sequence of joins and the size of the result relation for each join can be the product of the sizes of its operand relations in the worst case, the size of the result relation after several

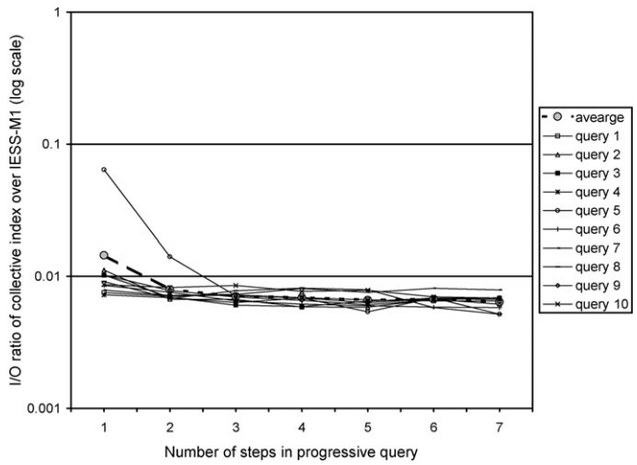

FIGURE 9. Performance comparison between collective index and IESS-M1.

steps could easily blow up. To avoid such a case, we chose external relations of size 1000 for the progressive queries in the experiments. The joining attribute pairs in the step-queries were randomly chosen among those whose result relation sizes are within the allowed range, as mentioned previously. Note that, since each tuple in an external relation in our experiments occupies one disk block, an index is beneficial for a query with a small selectivity even if the underlying relation is not very large. From the two figures, we have the following observations:

- Our collective index technique always outperforms IESS-M1 in terms of the number of I/Os accessed after each step for a multiple-input linear progressive query. Since the result relation size of a step-query does not monotonically decrease, the improvement achieved by the collective index technique does not monotonically decrease as more steps are processed.

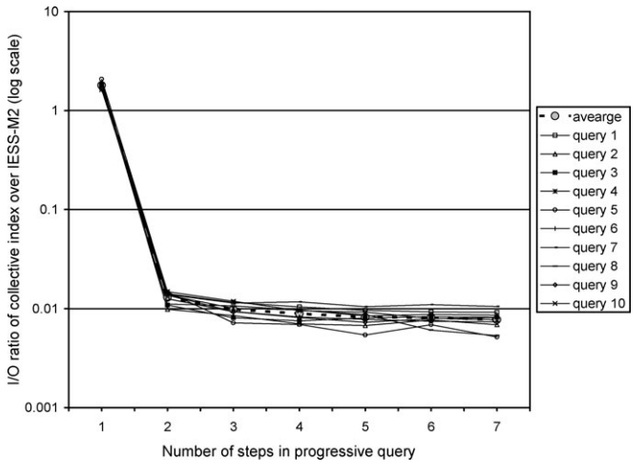

FIGURE 10. Performance comparison between collective index and IESS-M2.

- Our collective index technique outperforms IESS-M2 except for the first step. The reason why our technique does not win for the first step is the same as that for single-input linear progressive queries. That is, the collective index needs some overhead to maintain the router entries, while IESS-M2 can take advantage of a conventional B^+-tree without maintenance overhead.
- Our collective index technique can achieve much greater improvement for multiple-input linear progressive queries than for single-input linear progressive queries. This is because the index-based join method can improve the performance of the nested-loop join method for a (join) step-query much more significantly than the index-based scan method can do for a (unary) step-query over the sequential scan method. This result is encouraging since improving the performance of join queries is crucial in achieving an efficient database management system.

As mentioned earlier, among the conventional query processing strategies (REQM, IEDCI and IESS) that could be applied to process progressive queries, only IESS is practically feasible. We compared our collective index technique with variations (IESS-S1, IESS-S2, IESS-M1 and IESS-M2) of IESS for processing different types of progressive queries in our experiments. The empirical evaluation demonstrates that our collective index technique is quite promising in supporting efficient processing of progressive queries, comparing with the feasible conventional methods. The experiment results also show that our buffering strategies are beneficial.

6. CONCLUSIONS

There is an increasing demand for processing advanced types of queries in emerging application domains. In this paper, we have studied a new type of query for data-intensive applications, called progressive queries. Our main contributions are summarized as follows:

- We have introduced a query model to characterize different types of progressive queries based on their structures (linear and non-linear) and input requirements for their step-queries.
- We have proposed a new index structure, called the collective index, to efficiently process progressive queries. The collective index technique incrementally evaluates step-queries via dynamically maintained member indexes. Utilizing the special structure of the collective index, the (member) indexes on the input relation(s) of a step-query are efficiently transformed into indexes on the result relation.
- We have presented two algorithms for processing single-input linear and multiple-input linear progressive queries, respectively, based on the collective index.

Effective buffering (including bitmap) strategies to minimize I/O cost are also suggested.
- We have conducted an extensive experimental study to assess the performance of the collective index technique. Our experiment results show that the proposed technique generally outperforms the conventional query processing approaches such as the sequential scan (for unary queries) and the nest-loop join (for join queries) in processing progressive queries.

Our future work includes developing techniques for processing progressive queries of Type 3 and the ones involving aggregate functions. We will also explore issues for incorporating progressive query processing in a real database management system. In addition, we will investigate approximate progressive queries.

FUNDING

Research was partially supported by the US National Science Foundation (under grants #CNS-0521142, #IIS-0414594 and #IIS-0414576) and The University of Michigan (under UMD-CEEP and UM-OVPR grants).

REFERENCES

[1] Franklin, M., Halevy, A. and Maier, D. (2005) From databases to dataspaces: a new abstraction for information management. *SIGMOD Rec*., **34**, 27–33.

[2] Nieto-Santisteban, M.A., Gray, J., Szalay, A.S., Annis, J., Thakar, A.R. and O'Mullane, W. (2005) When Database Systems Meet the Grid. *Proc. CIDR Conf*., Asilomar, CA, USA, January 4–7, pp. 154–161, online proceedings, http://www-db.cs.wisc.edu/cidr/.

[3] Gray, J. and Szalay, A.S. (2004) Where the rubber meets the sky: bridging the gap between databases and science. *IEEE Data Eng. Bull*., **27**, 3–11.

[4] Stevens, R. *et al*. (2004) myGrid and the drug discovery process. *Drug Discovery Today: BIOSILICO*, **2**, 140–148.

[5] Nambiar, U., Lud, B., Lin, K. and Baru, C. (2006) The GEON Portal: Accelerating Knowledge Discovery in the Geosciences. *Proc. ACM Int. Workshop on Web Information and Data Management (WIDM)*, Arlington, Virginia, USA, November 4–7, pp. 83–90, ACM.

[6] Comer, D. (1979) The ubiquitous B-tree. *ACM Comput. Surv.*, **11**, 121–137.

[7] Ramakrishnan, R. and Gehrke, J. (2002) *Database Management Systems*. McGraw-Hill, New York.

[8] Calvanese, D., Giacomo, G.D., Lenserini, M. and Vardi, M.Y. (2005) View-Based Query Process: On the Relationship Between Rewriting, Answering and Losslessness. *Proc. ICDT Conf.*, Edinburgh, UK, January 5–7 pp. 321–336, Springer.

[9] Gou, G., Kormilitsin, M. and Chirkova, R. (2006) Query Evaluation Using Overlapping Views: Completeness and Efficiency. *Proc. SIGMOD Conf.*, Chicago, Illinois, USA, June 27–29, pp. 37–48, ACM.

[10] Halevy, A.Y. (2001) Answering queries using views: a survey. *VLDB J*., **10**, 270–294.

[11] Mistry, H., Roy, P., Sudarshan, S. and Ramamritham, K. (2001) Materialized View Selection and Maintenance Using Multi-Query Optimization. *Proc. SIGMOD Conf*., Santa Barbara, California, May 21–24, pp. 301–318, ACM.

[12] Agarwal, P.K., Xie, J., Yang, J. and Yu, H. (2006) Scalable Continuous Query Processing by Tracking Hotspots. *Proc. VLDB Conf*., Seoul, Korea, September 12–15, pp. 31–42, ACM.

[13] Babu, S. (2005) Adaptive query processing in data stream management systems. PhD Dissertation, Stanford University.

[14] Lim, H.-S., Lee, L.-G., Lee, M.-J., Whang, K.-Y. and Song, I.-Y. (2006) Continuous Query Processing in Data Streams Using Duality of Data and Queries. *Proc. SIGMOD Conf*., Chicago, Illinois, June 27–29, pp. 313–324, ACM.

[15] Mokbel, M.F. (2004) Continuous Query Processing in Spatio-Temporal Databases. *Proc. EDBT Workshops*, Heraklion, Crete, Greece, March 14–18, pp. 100–111, Springer.

[16] Antoshenkov, G. (1993) Dynamic Query Optimization in RDB/VMS. *Proc. IEEE ICDE Conf*., April 19–23, pp. 538–547, IEEE Computer Society.

[17] Babu, S. and Bizarro, P. (2005) Adaptive Query Processing in the Looking Glass. *Proc. CIDR Conf*., Asilomar, CA, USA, January 4–7, pp. 238–249, online proceedings, http://www-db.cs.wisc.edu/cidr/.

[18] Kabra, N. and DeWitt, D.J. (1998) Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. *Proc. SIGMOD Conf*., Seattle, Washington, USA, June 2–4, pp. 106–117, ACM.

[19] Liu, L. and Pu, C. (1997) Dynamic query processing in DIOM. *IEEE Data Eng. Bull*., **20**, 30–37.

[20] Lu, H., Tan, K.-L. and Dao, S. (1995) The Fittest Survives: An Adaptive Approach to Query Optimization. *Proc. VLDB Conf*., Zurich, Switzerland, September 11–15, pp. 251–262, Morgan Kaufmann.

[21] Jarke, M. and Koch, J. (1984) Query optimization in database systems. *ACM Comput. Surv*., **16**, 111–152.

[22] Graefe, G. (1993) Query evaluation techniques for large databases. *ACM Comput. Surv*., **25**, 73–123.

[23] Silberschatz, A., Korth, H.F. and Sudarshan, S. (2005) *Database System Concepts*, (5th ed). McGraw-Hill, New York.

[24] Zhu, Q. and Larson, P.-Å. (1998) Solving local cost estimation problem for global query optimization in multidatabase systems. *Distrib. Parallel Databases*, **6**, 373–420.

[25] Babu, S. and Widom, J. (2001) Continuous queries over data streams. *SIGMOD Rec*., **30**, 109–120.