Journal of Computer and System Sciences 79 (2013) 330–348

Contents lists available at SciVerse ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Optimizing adaptive multi-route query processing via time-partitioned indices

Karen Works *, Elke A. Rundensteiner, Emmanuel Agu

Worcester Polytechnic Institute, Worcester, MA, United States

ARTICLE INFO

Article history:
Received 5 May 2011
Received in revised form 20 January 2012
Accepted 11 September 2012
Available online 22 October 2012

Keywords:
Data stream database systems
Indexing
Adaptive query processing

ABSTRACT

Adaptive multi-route query processing (AMR) is an emerging paradigm for processing stream queries in highly fluctuating environments. The content of stream data can be unpredictable. Thus, instead of selecting a fixed plan, AMR dynamically routes batches of tuples to operators in the query network based on up-to-date system statistics. The workload of query access patterns in AMR systems is ever changing. Selecting a single best index may not efficiently support all query access patterns at all times. While maintaining multiple indices to match a variety of query access patterns increases overhead and decreases throughput. Index design, while paramount for efficient query execution, is particularly challenging in AMR systems as the indices must serve the continuously evolving query access patterns. Our proposed Adaptive Multi-Route Index (AMRI) employs a bitmap time-partitioned design that serves a diverse ever changing workload of query access patterns and remains lightweight in terms of maintenance and storage requirements. We propose a high quality yet efficient assessment method modeled after hierarchical heavy hitters that exploits route relationships by modeling the frequency of the search access patterns used as nodes in a lattice. We also design assessment scheduling methods for AMRI based upon detecting changes in the search access patterns used. Our AMRI incorporates migration strategies that seek to meet the needs of both old partially serviced and new incoming search requests. Our experimental study using both synthetic and real data streams demonstrates that AMRI strikes a balance between effectively supporting dynamic stream environments while keeping the index overhead to a minimum. Using an environmental data set collected in the Intel Berkeley Research lab, our AMRI produced on average 68% more cumulative throughput than the state-of-the-art approach.

1. Introduction

1.1. Background on index tuning

Abadi et. al. [1] predicted the increase in the number of monitoring applications. Many monitoring applications were once simple (i.e., a simple query). They have become increasingly more complex (i.e., multiple complex queries that share data). Consider the stock market. A few years ago, a market analyst may only have assessed the current stock price and volume. Today, to track rapid market shifts, the same analyst has readily access to many more up-to-date statistics on the latest company and market sector information from news feeds, web sites, and blogs. In short, monitoring applications increasingly require complex queries integrating may stream sources.

* Corresponding author.
E-mail addresses: kworks@cs.wpi.edu (K. Works), rundenst@cs.wpi.edu (E.A. Rundensteiner), emmanuel@cs.wpi.edu (E. Agu).

0022-0000/$ – see front matter
http://dx.doi.org/10.1016/j.jcss.2012.09.011

Such complex queries must efficiently process over long durations with possibly frequent data fluctuations [2]. Fluctuations cause periodic variances in the selectivity and performance of operators. This can render the most carefully chosen plan sub-optimal and possibly ineffective [3]. To address this, much research has been done on data stream management systems (DSMS) that continuously adjust the query path along which incoming tuple are processed [3–7]. We refer to these as Adaptive Multi-Route systems (AMR).

AMRs adapt the execution order of operators (i.e., the query path) to current statistics. The prominent AMR, Eddy [3], utilizes a central routing operator. Given statistics, it decides for each tuple which operator to visit next. The query path taken thus far by tuple t_i determines which tuples from the different incoming streams, tuple t_i will be composed of. When t_i is processed by join operator op_j, all the tuples that compose t_i are used to generate the criteria to locate tuples stored in op_j's state that should join with t_i. Tuples may take different query paths and thus can be composed of tuples from different streams. Thus they will generate diverse search criteria that must be efficiently supported by join operators. Given the complexity of monitoring application queries, current DSMS cannot create a separate index for all possible search criteria. Worst yet, when path fluctuations occur, AMRs must quickly and precisely adjust the index configurations in use to best serve the new path(s). This challenging problem, ignored by the literature thus far, is the focus of our work.

1.2. AMR index requirements

Traditional off-line index tuning approaches select the "best" fixed index structure off-line [8–11]. They do not meet the needs of AMRs where the search request workload is in constant flux. AMRs require an online index tuning solution. Such approaches continuously evaluate and adjust the index structure online via the index tuning life cycle [12–15]. The steps in the life cycle are to: 1) identify changes in the search request workload (*assessment*), 2) locate the "best" index structure (*selection*), and 3) re-index states using the current "best" index structure (*migration*).

Unlike indexing in static database systems, AMRs face several unique challenges. 1) Periodically the router sends tuples to suboptimal operators to update statistics [3]. The access patterns generated by such tuples often have low frequencies. The frequency statistics from such tuples often have little influence on the final index structure selected. However, they add additional overhead to both index assessment and selection. To reduce such overhead, AMRs must adjust the quality and quantity of statistics collected. If too few statistics are kept the overhead may be reduced but a suboptimal index configuration may be found. If too detailed statistics are kept, the optimal index may be located but it may require too high an overhead. 2) AMRs must continuously assess and adjust indices to best support the query paths currently in use. The abruptness and frequency of such changes make this extremely challenging. Furthermore the overhead of assessing indices must not detract from producing rapid results. 3) When the query paths used to process incoming tuples evolve, the system will contain old partially serviced tuples processed along query paths different from the new "best" query paths. To minimize response times AMRs must efficiently process both old partially serviced and new incoming tuples.

1.3. The proposed approach: AMRI

In short, AMRs require an index design that: 1) supports diverse search criteria, 2) requires minimal maintenance, 3) compact so to fit in main memory, and 4) can efficiently process all search criteria.

AMRs require index tuning, following the life cycle, that: 1) efficiently and accurately identifies abrupt key query path changes while not detracting from the production of rapid results (*assessment*), 2) maintains the quality of the best index configuration selected within a preset threshold (*selection*), and 3) determines the best approach to re-indexing stored tuples to efficiently process the old partially serviced tuples (*migration*). Our proposed *Adaptive Multi-Route Index* (*AMRI*) solution incorporates a physical index design and customized index tuning methods that meet the above requirements.

Our preliminary ideas on AMR index assessment was published in a workshop proceeding [16]. In this work, we focused on the design of two assessment methods called *Compact Self Reliant Index Assessment CSRIA*, and *Compact Dependent Index Assessment CDIA*. Both reduce assessment overhead by compacting the statistics collected. *CSRIA* utilizes a heavy hitter method [17]. *CDIA* tracks statistics in a lattice, exploiting the dependent relationships between the search criteria, and employs a hierarchical heavy hitter method [18].

In this paper, we extend our basic index assessment to efficiently support the index tuning life cycle by incorporating scheduling assessment methods as well as efficient index migration methods. We also establish the bounds on the optimality (i.e., quality) of the index configuration found during selection using the compact statistics generated by our *CSRIA* and *CDIA* assessment methods.

Our new contributions include:

1) We enhance our index design to support both old partially serviced and new incoming search requests by employing a time-partitioned index structure.
2) We formulate and prove that the effect of statistic compaction on the potential loss of quality of the selected index can be bounded by a preset constant.
3) We propose methods for scheduling index tuning. In particular, our *Triggered* method schedules assessment on states with observed routing shifts.

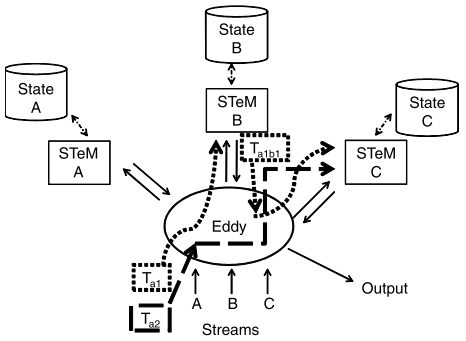

Fig. 1. Query Q1 in the AMR system.

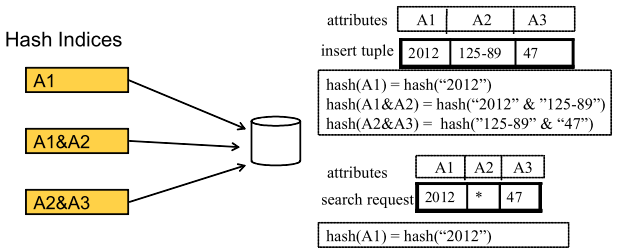

Fig. 2. Indexing in AMR systems.

4) We propose a *Partial Migration* strategy that migrates only some stored tuples based on their estimated future use. Through our study of migration approaches we explore the question of whether or not it is effective to maintain an index configuration for subsets of stored tuples.
5) We conduct extensive experiments (beyond those in [16]) to evaluate the effectiveness of our proposed scheduling and migration methods using both real and synthetic data over a large variety of experimental conditions including varying the amount of fluctuations in the query paths used. We demonstrate that *AMRI* always wins over the current AMR index approaches, including hash indices [5] and bitmap indices [19].

2. Motivation

2.1. Example of search requests with varying criteria

Query Q1:
SELECT * **FROM** StreamA, StreamB, StreamC
WHERE StreamA.A1 = StreamB.A1 **AND** StreamB.A3 = StreamC.A3 **AND** StreamC.A2 = StreamA.A2
WINDOW 30 *seconds*

Example. Consider query Q1, the AMR in Fig. 1, and tuples t_{a1} and t_{a2} from *StreamA*. t_{a1} is first routed to join with tuples from *StreamB* in STeM B. Then resulting tuple t_{a1b1} is routed to join with tuples from *StreamC* in STeM C. While t_{a2} is instead first routed to join with tuples from *StreamC* in STeM C. STeM C must be able to efficiently locate tuples from *StreamC* using different search criteria. In particular, for tuple t_{a2} only the join attributes between *StreamA* and *StreamC* are used, i.e., search on attribute $A2$. While for tuple t_{a1b1} the combined join attributes of *StreamA* and *StreamB* and *StreamC* are used, i.e., search on both attributes $A2$ and $A3$.

2.2. State-of-the-art indexing in AMR systems

The main work on improving the precision of indices in AMRs is by Raman et al. [5]. They proposed to create a state for each stream to store its incoming tuples (i.e., akin to traditional database tables) and multiple hash indices for each state. Each hash index of the join operator (i.e., STeM) optimizes a specific data access on a state. We now illustrate the inefficiencies of such indices.

Example. Consider a DSMS that tracks location of packages via sensors. Each sensor propagates 3 attributes: *priority code* ($A1$), *package id* ($A2$), and *location Id* ($A3$). The state storing the sensor data has 3 hash indices, i.e., $A1$, $A1 \& A2$, and $A2 \& A3$ (Fig. 2). To insert tuple t_i, first t_i is stored in the state. Then hash keys for $t_i.A1$, $t_i.A1 \& t_i.A3$, and $t_i.A2 \& t_i.A3$ are created, linked to t_i, and stored.

Select	<agg-func-list>
From	<stream-name>
Where	<preds>
Window	<***window-length***> : *default-window-length*

Select	A.*, B.*, C.*
From	StreamA A, StreamB B, StreamC C
Where	A. A1 = B. A2 and A. A3 = C. A3 and B. A3 = C. A4
Window	10000

Fig. 3. SPJ query template and example.

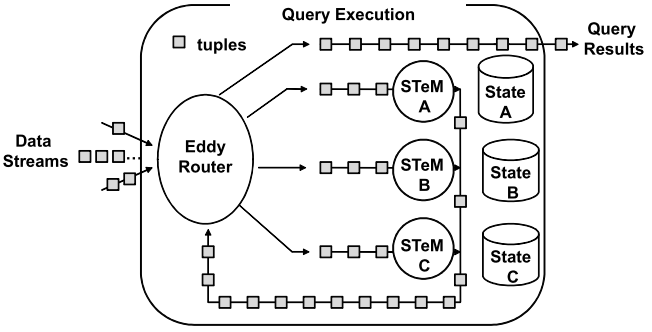

Fig. 4. AMR of example.

Consider search request sr_1 that looks for packages with priority code $A1 = 2012$ and location id $A3 =$ '47'. To execute sr_1, first, which hash index is most suitable to process sr_1 is located. The most suitable index is the hash index that supports the largest number of attributes in sr_1 and contains no attributes not in sr_1. Next a lookup is performed using the most suitable index, e.g., $A1$.

Consider sr_2 that looks for packages where location id $A3 =$ '47'. No suitable index exists for sr_2. The entire state is scanned. To improve the search time of sr_2 requires the creation of a new hash index to support attribute $A3$. However this adds additional memory and maintenance costs to store tuples in the state.

Unfortunately to support diverse criteria as motivated above requires: 1) a possibly large number of hash indices, 2) high memory overhead to support multiple links for each stored tuple, and 3) considerable maintenance costs to create and delete multiple hash indices. Our experimental study (Section 9) confirms these inefficiencies and demonstrates that our proposed solution successfully overcomes these shortcomings.

3. Preliminaries

Query model. We consider SPJ (select-project-join) queries processed under a *suffix* window (Fig. 3). A suffix window indicates the length of the data history to be queried via sliding windows. Our explanation uses one SPJ query. Yet, our AMRI equally applies to multiple SPJ queries.

A *state* is instantiated for each stream in the FROM clause, e.g., a state for *StreamA*, *StreamB*, and *StreamC* (Fig. 4 is an AMR for the query above). Each state is associated with a unary join *STeM operator* [5] that supports insertion and deletion of tuples, and locating tuples based on join predicates.

Join operator. The original AMR system, Eddy [3], utilized ripple join operators in which the intermediate results produced are stored in the operators' state. [20] improved query execution time of AMR systems by introducing the STAIRS operator which is an extension to the ripple join operator. Beyond the standard join processing, the STAIRS operator allows the system to dynamically undo some of the query processing performed on blocked tuples, i.e, such tuples are waiting upon intermediate results from other streams. This allows the STAIRS operator to unblock such tuples.

Both the ripple join and STAIRS operators have the burden of maintaining the probe functionality of tuples from the same stream in multiple operators [5]. In addition, their route selection suffers as the router has no knowledge about what is affecting an operator's selectivity. Namely, it cannot be determined if the probes that do or do not utilize the cached results are more selective [5].

To address this, Rama et al. improved optimization capabilities by extending routing flexibility through their STeM operator [5]. A STeM is a unary join operator dedicated to the storage and probing of requests containing tuples from one base stream. Given the promise of this strategy, we adopt SteMs as our join operator, though other types of joins could equally be used.

A *join predicate* is expressed in the query WHERE clause composed of 1) an attribute stream reference, 2) a join expression ($=$, $<$, $>$ $\geqslant$, $\leqslant$), and 3) another attribute stream reference (e.g., $A.A1 = B.A2$). The *join attribute set JAS* for a state (i.e.,

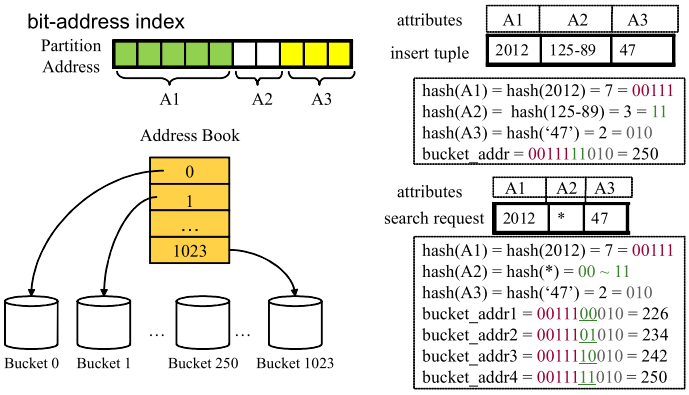

Fig. 5. State using a bit address index.

say from *StreamA*) is the set of all attributes specified in the join predicates (i.e., for *StreamA* $JAS = \{A1, A3\}$). Each STeM can search for tuples stored in its state based upon any combination of attributes in *JAS*. For example, the combinations of attributes in *JAS* for *StreamA* are $A1$, $A3$, and $A1 \,\&\, A3$ combined. An *access pattern* (*ap*) is an attribute combination used to specify search criteria. It is denoted by a vector whose size equals the cardinality of the *JAS*. Each vector position represents one join attribute. Join attributes used to search are represented by the name of the attribute. Those not used are represented by the wild card symbol $*$.

Example. Reconsider the state of *StreamA* where $JAS = \{A1, A3\}$. $\langle A1, A3\rangle$ is an *ap* where all join attributes are used to search. While $\langle A1, *\rangle$ is an *ap* only using join attribute $A1$, and $\langle *, *\rangle$ is an *ap* using no join attributes (i.e., a full scan).

4. Adaptive multi-route index

4.1. Physical index design

We now explain our proposal to employ a versatile yet compact bit-address index as the foundation of our solution [19]. Our bit-address index incorporates an index key map (a.k.a. the *index configuration IC*) or blueprint to the memory location where tuples are stored. Each *IC* of B bits supports 2^B buckets to store tuples. The bucket in which tuple t_i is stored is found by using the *IC* to map t_i's attribute values to a specific bucket. *IC* delineates for each join attribute the number of bits (possibly none) used in the mapping. No index links are ever created as the *IC* derived for each tuple is never stored. This significantly reduces the memory and CPU overhead.

Example. Reconsider the example in Section 2.2. Tuple t_i, a *package* record, is inserted into bit-address index *BI* in Fig. 5 where the *IC* has 10 bits (5 bits for attribute $A1$, 2 bits for $A2$, and 3 bits for $A3$). First the bucket id for t_i is generated by mapping the values for $t_i.A1$, $t_i.A2$, and $t_i.A3$, which are 00111, 11, and 010 respectively. Then these values are combined to form the bucket id, i.e., 0011111010 or bucket 250. Finally, t_i is stored in bucket 250. The *IC* generated is not stored. Thus, no memory or CPU time is required to support index links. In contrast, the hash index approach [5] (Section 2.2) utilizes both memory and CPU time to create keys for each hash index created. Thus, the bit-address index satisfies the low memory and CPU requirement.

As we will now show, adapting a bit-address index requires on average less CPU time than the hash index approach. To adapt tuples in the state from index BI_1 to BI_2 requires the relocation of stored tuples to the buckets defined by BI_2, and deletion of BI_1's memory. While the traditional hash index approach would create and delete multiple hash keys for each stored tuple.

Consider the search performance using the *BI* in Fig. 5 versus the hash indices in Fig. 2. Search request sr_1 is looking for all packages with priority code = 2012 and location id = '47' (Fig. 5). First the bucket ids to be searched are found by mapping the attributes specified sr_1 (i.e., $A1 = 2012$ and $A3 =$ '47' or 00111 and 010) and the attributes not specified in sr_1 (i.e., all possible biit values for $A2$ or 00 to 11). Then the possible bits are combined to identify the buckets (i.e., 0011100010, 0011101010, 0011110010, and 0011111010 or buckets 226, 234, 242, and 250). Finally a scan is performed across all identified bucket(s). If the search is narrow and precise (i.e., the access pattern specifies all join attributes), then only one bucket is searched. If the search is wide (i.e., the access pattern contains wild card symbols) then several buckets are searched. On average, a good index configuration limits the number of buckets examined for the majority of search requests. Clearly, a single *BI* can serve multiple diverse access patterns and require less maintenance overhead than multiple hash keys. Thus, the bit-address index satisfies the diverse access pattern requirement (Section 1.3).

Table 1
Notations.

Notation	Meaning
ap	a search access pattern
λ_d	# of incoming tuples from a stream received within a time unit
λ_r	# of search requests received within a time unit
C_h	average cost for computing a hash function
C_c	average cost for conducting a value comparison
C_d	average cost to delete a value from a bucket
N_A	# of indexed attributes
$N_{AP,ap}$	# of indexed attributes specified in ap
W_{ap}	window length (in # of time units) of ap
B_{ap}	# of bits assigned to all attr. specified in ap
F_{ap}	frequency of ap
AP	the set of access patterns in workload D

The configuration of the index key map (i.e., the assignment of which attribute data values map to which bits) influences the number of buckets searched. The optimal index key map is configured so that there exists a relatively even distribution of stored tuples in each bucket. This requires the analysis of the distributions of the data content of each index attribute.

4.2. Multiple index configurations

In AMR systems, the query paths used to process tuples constantly evolve. As they evolve, the system will contain some partially serviced tuples already in process. These partially serviced tuples may favor different query paths than new incoming tuples. To minimize response time, states must efficiently process search requests generated by all tuples. To support this search request diversity, we propose to allow states to maintain multiple index configurations. Namely, a distinct index configuration for different subsets of tuples.

Tuple t_i will join with stored tuples that are within the query window of t_i. For this, we propose to time-partition stored tuples by the windows of tuples that they produce results for. Each partition of stored tuples is distinctly indexed to support efficient processing of the search requests that they provide results for, e.g., indexed to support the query paths of older tuples.

Partition design. Each index configuration chosen is the most efficient for the query windows into which the stored tuple(s) under it fall. Stored tuples are broken into time slices referred to as partitions P. A partition is a portion of incoming tuples of constant length $|P|$, e.g., based upon tuple count.

Each partition is assigned a partition number. The current partition number is equal to $\lfloor \frac{\lambda_d}{|P|} \rfloor$ where λ_d is equal to the number of incoming tuples received thus far from a stream. A query window QW is composed of a number of partitions denoted by PW, where $PW = \lceil \frac{QW}{|P|} \rceil$. Each configuration is stored in a hash table via a partition number key. To reduce overhead, consecutive partitions that use the same configuration are stored in the same BI.

Processing costs using partitions. We now outline the costs of insertion, search, and deletion of tuples using AMRI.

Insertion: To store tuple t_i in a state the proper bucket within the latest partition must be located. The complexity of insertion is constant, namely, $O(C_{hash,I} + C_{insert})$. Notations are in Table 1.

Search: Given search request sr_1, first the set of partitions in the query window relevant to sr_1 referred to as $QW(sr_1)$ are determined. Next all bit-address indices in the partitions of $QW(sr_1)$ referred to as $BI_{QW(sr_1)}$ are located. Finally all tuples that match sr_1 stored in buckets located by the indices in $BI_{QW(sr_1)}$ are found. In the worst case a unique index exists for each partition. Then the complexity of search is $O(|BI_{QW(sr_1)}|(C_{hash,sr_1} + C_{search}))$. In the best case all partitions share the same index and the complexity of search is simply $O(C_{hash,sr_1} + C_{search})$.

Deletion: AMR systems do not force a strict FIFO processing order by tuple arrival time so to allow tuples to travel through the AMR concurrently along their respective query paths [3]. This disorder causes deletion to be more complicated since a stored tuple should only be removed when no search request exists that could potentially still join with it. To assure correctness inspite of this disorder, the router tracks the number of search tuples by partition number and the earliest partition number of all search tuples, denoted as EPN. Only tuples from partitions older than the oldest partition in the query window of EPN are removed. The complexity of deletion is $O(N_{PR}(PW)C_d)$ *where* $N_{PR} =$ *# of partitions to remove and* $PW =$ *# of tuples in a partition.*

As seen above partitioning is a well established method in streaming to reduce cost of state maintenance by performing some operations (e.g., purging) at a batch rather than individual tuple level. Similarly, we hence use partitioning here in

AMRI although it is not a key factor in the overhead of AMRI. Partitions store a portion of the incoming tuples from a stream, i.e., the partitioning size is set to be a portion of the window size. Each partition will produce results for a particular subset of tuples bounded by arrival time or tuple count. Partitioning reduces deletion overhead. When it is determined that no more tuples that will join with a partition are still processing in the system, the entire partition worth of tuples can simply be deleted. In contrast, non-partitioned states must periodically search through the state and compare individual tuples to the current window and accordingly delete. Hence partitions support cheaper deletions as they allow batch comparisons and deletions.

Compared to non-partitioned indices there is a minimal additional search overhead for partitioned indices, namely, looking up the index configuration used by each partition. However, adjacent partitions use the same index if they share the same index configuration. Thus in the worst case (i.e., where a unique index configuration exists for each partition) the search requires a look up of the index configuration of each partition. In the best case, the search simply requires a single index configuration look up.

5. Index assessment

The *assessment* component maintains compact statistics used to locate the optimal index configuration for each state, i.e., the index configuration with the lowest index configuration dependent costs C_D.

5.1. Index configuration dependent costs C_D

The quality of an index configuration *IC* depends upon the estimated *unit processing costs* for *IC* [21] which is the combined maintenance and search costs [19]. The maintenance costs are the sum of the cost to insert and delete tuples in a state and to compute bucket ids ($C_{insert} + C_{delete} + C_{hash,I}$) (Table 1). The search costs are the sum of the cost to compute bucket ids and search for tuples stored in the state ($C_{hash,SR} + C_{search}$). Both C_{insert} and C_{delete} are independent of which *IC* is evaluated. Henceforth when comparing index configurations we only consider the index configuration dependent costs C_D (Eq. (1)) [19]. The insertion cost to compute bucket id $C_{hash,I}$ is equal to the product of the number of incoming tuples, the number of attributes, and the cost to compute the key for one attribute or $\lambda_d N_A C_h$. The search cost to compute bucket ids $C_{hash,SR}$ is equal to the product of the number of incoming search requests and the sum of the cost to compute the key for the attributes in *AP* or $\lambda_r \sum_{ap \in AP} N_{AP,ap} C_h$. The search cost C_{search} is equal to the product of the number of incoming search requests and the estimated number of tuples scanned or $\lambda_r \sum_{ap \in AP} \frac{\lambda_d W_{ap} F_{ap}}{2^{B_{ap}}} C_c$.

$$C_D = C_{hash,I} + C_{hash,SR} + C_{search}$$
$$C_D = \lambda_d N_A C_h + \lambda_r \sum_{ap \in AP} N_{AP,ap} C_h + \lambda_r \sum_{ap \in AP} \frac{\lambda_d W_{ap} F_{ap}}{2^{B_{ap}}} C_c \quad (1)$$

5.2. Access pattern statistics

To assess *IC*s requires the frequency statistics of each access pattern (F_{ap}). Recall that an access pattern does not correspond to raw stream data tuples but rather a search criteria (Section 3).

Definition 1. The *frequency of access pattern ap* in a workload D arising during a given time duration (namely, all incoming and intermediate tuples processed over that time duration), denoted as F_{ap}, is $F_{ap} = \frac{N_{ap}}{|D_{AP}|}$ where N_{ap} corresponds to the cumulative number of times access pattern *ap* occurs in D and $|D_{AP}|$ is the total number of access patterns in D.

The number of possible access patterns is equal to the number of combinations of join attributes for a state. If there are N_{ja} join attributes then the number of possible access patterns is $\sum_{k=1}^{N_{ja}} \binom{N_{ja}}{k}$ or $2^{N_{ja}} - 1$. Thus the number of possible access patterns is exponential in the number of join attributes.

5.3. Self reliant index assessment SRIA

5.3.1. Basic SRIA

We first sketch *Self Reliant Index Assessment* or SRIA. SRIA captures the frequency of access patterns for a state in a hash table referred to as the *SRIA* table. Access pattern statistics collected in the *SRIA* table are independent of each other (i.e., self reliant).

To allow quick direct referencing to every access pattern *ap* in the *SRIA* table, each *ap* is mapped to a unique binary representation $B(ap)$. $A1$ indicates that a join attribute is used to search, while a 0 indicates that a join attribute is *not*. Consider a state with 3 join attributes $\{A1, A2, A3\}$. If ap_1 is searching using only attribute A (i.e., $ap_1 = \langle A1, *, * \rangle$) then

Table 2
Compact SRIA estimation example.

ap	F_{ap}	ap	F_{ap}
$\langle A1, *, * \rangle$	4%	$\langle A1, A2, * \rangle$	4%
$\langle *, A2, * \rangle$	10%	$\langle A1, *, A3 \rangle$	16%
$\langle *, *, A3 \rangle$	10%	$\langle *, A2, A3 \rangle$	10%
		$\langle A1, A2, A3 \rangle$	46%

$B(ap_1) = 100$ which represents index number 4. If ap_1 is searching using attributes $A2$ and $A3$ (i.e., $ap_1 = \langle *, A2, A3 \rangle$) then $B(ap_1) = 011$ which represents index number 3.

For each incoming access pattern ap the binary representation $B(ap)$ is found. If the access pattern statistic exists then N_{ap} is incremented by 1. Otherwise the new access pattern ap is added with N_{ap} initialized to 1.

Definition 2. Given a state S_t, the ***SRIA* algorithm** outputs the set of all frequency access patterns collected for S_t.

5.3.2. Compact SRIA: access pattern reduction

The large number of possible access patterns that is exponential in the number of join attributes N_{ja} (Section 5.2) can cause potential memory limitations. We now explore a frequency access pattern reduction method extension to *SRIA* referred to as *Compact SRIA* or *CSRIA* is modeled after the heavy hitter algorithm proposed by Manku and Motwani [17]. Unlike [17], we compact and collect statistics on search request frequencies instead of raw data tuples. Informally, during assessment *CSRIA* periodically removes any access pattern statistic whose frequency falls below a preset error rate ϵ. Upon completion of assessment, *CSRIA* returns all access pattern statistics whose frequencies are above a preset threshold θ.

Definition 3. Given a state S_t, SRIA access patterns for S_t, threshold θ, maximum error in the F_{ap} collected δ, and error rate ϵ, the **CSRIA algorithm** outputs the set of frequency access patterns Q such that: $\forall ap \in Q\colon (F_{ap} + \delta) \geqslant (\theta - \epsilon)$ and $\forall ap \in (N_{ap} - Q)\colon (F_{ap} + \delta) < (\theta - \epsilon)$.

CSRIA evaluates incoming access patterns in segments. Each segment contains $\lceil \frac{1}{\epsilon} \rceil$ search requests. Segments are associated with an id (s_{id}) that represents the current required number of search requests for an access pattern to meet the preset error rate threshold. The current segment id is equal to $\lfloor \epsilon * \lambda_r \rfloor$ where λ_r is equal to the number of search requests received during assessment thus far. To ensure that certain access pattern statistics are not deleted too early, the current maximum frequency error δ is stored with each access pattern statistic. The maximum frequency error δ represents the minimum number of requests that should occur in order for frequency access pattern F_{ap} to be above the preset error rate threshold.

CSRIA is composed of three phases, *insertion* (creating access pattern statistics), *compression* (removing access pattern statistics), and *final results* (finding all access pattern statistics whose frequency meets the threshold). *Insertion* and *compression* occur during the assessment phase. During assessment, creating access pattern statistics from the access patterns of incoming search requests (*insertion*) proceeds as follows: First, the binary representation $B(ap)$ is computed. Second, if the access pattern statistic exists then N_{ap} is increased by 1. Otherwise a new access pattern statistic is added with N_{ap} equal to 1 and the maximum frequency error δ equal to one less than the current required number of search requests for access pattern ap to meet the preset error rate threshold, i.e., $s_{id} - 1$. Also during assessment, *compression* is executed whenever a segment worth of search requests are received. *Compression* removes any access pattern from the SRIA table whose frequency is below the preset error threshold, i.e., $N_{ap} + \delta \leqslant s_{id}$. At the end of assessment, the *final result* produced corresponds to all access patterns whose $F_{ap} + \delta$ is greater than the preset threshold θ in accordance with the maximum frequency error ϵ, i.e., $F_{ap} + \delta \geqslant \theta - \epsilon$.

CSRIA benefits. 1) It guarantees to find any access pattern whose frequency is greater than threshold θ. 2) The maximum memory required is limited to $\frac{1}{\epsilon} log(\epsilon \sum_{m=0}^{N_{ja}-1} \binom{N_{ja}-1}{m})$ access patterns where there are N_{ja} join attributes [22].

5.4. Dependent index assessment DIA

Example. As we will now demonstrate, CSRIA misses an opportunity for finding the optimal index configuration. CSRIA, while efficient, does not utilize the relationships between access patterns. This decreases the opportunity to find the optimal index configuration. Consider a state with 3 join attributes (SRIA Table 2) and a 4 bit *BI*. If θ is 5% and ϵ is .1%, then *CSRIA* will delete access patterns $\langle A1, *, * \rangle$ and $\langle A1, A2, * \rangle$ even though both would benefit from the index configuration $\langle A1, *, * \rangle$. Furthermore the combined frequency of $\langle A1, *, * \rangle$ and $\langle A1, A2, * \rangle$ is 8%, i.e., greater than θ. The index configuration found during selection has 1 bit assigned to the $A2$ attribute and 3 bits assigned to the $A3$ attribute. Whereas the optimal index configuration has 1 bit assigned to $A1$ and $A2$ attributes each and 2 bits assigned to the $A3$ attribute.

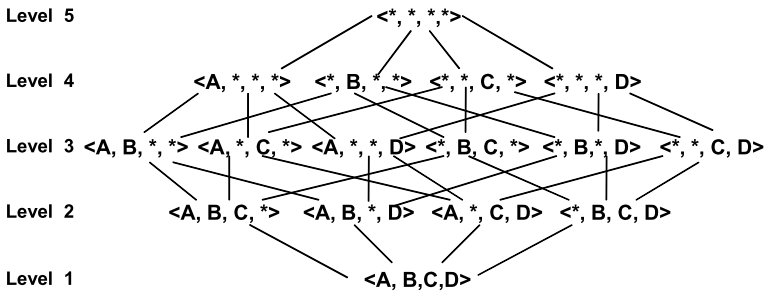

Fig. 6. DIA lattice for state with 4 join attributes.

5.4.1. Basic DIA

We now explore how the **dependent access pattern relationships** affect assessment. We first outline how a search request is executed based on the index configuration of a state. Then we describe how we exploit these relationships in *Dependent Index Assessment* or DIA.

The efficiency of executing a search request with access pattern ap_i depends upon the index configuration of the state. An index which only contains a subset or even all of the join attributes in ap_i will more efficiently support ap_i than an index that supports none of the join attributes in ap_i. Consider a state with $JAS = \{A1, A2, A3, A4\}$ and a search request with $ap = \langle A1, A2, *, * \rangle$:

Worst case. *IC* consists of no attributes in *ap* (e.g., *IC* consists of attributes *A*3 and *A*4). As *IC* and *ap* have no common attributes, no buckets are eliminated. Thus a full scan is required (i.e, a comparison of all stored tuples).

Slightly better case. *IC* consists of some attributes in *ap* and some not in *ap* (e.g., *IC* consists of attributes *A*1, *A*2, and *A*3). Each attribute in *IC* that is not in *ap* creates the search wild card condition (Section 4.1). If *n* is the number of bits assigned to the attributes not in *ap* then 2^n buckets must be compared.

Much better case. The attributes in *IC* are a subset of the attributes in *ap* (e.g., *IC* consists of attribute *A*1). In this case, only a single bucket must be searched. The bucket will contain all tuples relevant to *ap* as no wild card condition exists. However, not every tuple is guaranteed to satisfy *ap*. Overall, the number of tuples compared is likely smaller than the cases above.

Optimal case. The attributes in the *IC* exactly match the attributes in *ap* (e.g., *IC* consists of attributes *A*1 and *A*2). Since all attributes in *IC* are specified in *ap*, a single bucket will be searched. Further, all tuples in the bucket will correspond to matches. In other words, this results in the smallest number of tuples compared to all the cases above.

Definition 4. An index based upon access pattern ap_1 provides a **search benefit** to a search request utilizing access pattern ap_2, denoted as $ap_1 \prec ap_2$, if $\forall a \in ap_1 \rightarrow a \in ap_2$ where *a* is an attribute.

The search benefit relationships organize access patterns into a lattice which we refer to as the DIA lattice (Fig. 6). Each DIA lattice node emulates an access pattern. The DIA lattice starts with the access pattern with no join attributes (top node). At each DIA lattice level, nodes are formed by taking each node in the prior level and adding one join attribute not already in it. This process continues until all possible join attributes are included in the access pattern (bottom node). Each node is linked via an edge to the nodes it provides a *search benefit* to in the level directly below it.

Dependent Index Assessment DIA stores the assessment values in a DIA lattice to retain the dependent search benefit relationships between access patterns. DIA builds the DIA lattice in a top–down manner at runtime. Each access pattern node *nd* in the partial DIA lattice consists of the access pattern it represents, namely *nd.ap*, and the frequency of *nd.ap* requests, or $nd.F_{ap}$.

For each search request, if the access pattern node already exists in the DIA lattice then the corresponding frequency $nd.F_{ap}$ is incremented by 1. Otherwise, a new access pattern node is created. To enable quick direct referencing to each node, nodes are mapped to unique binary numbers as outlined above for *SRIA*. As such, physically each *DIA* node is stored in a *SRIA* table.

5.4.2. Compact DIA: access pattern compression

The large number of possible access patterns in DIA (Section 5.2) can cause memory limitations. We now explore an access pattern reduction method extension to *DIA* modeled after a hierarchical heavy hitter algorithm [18], namely *Compact DIA*, or *CDIA*. In our context, we utilize the search benefit relationship to combine access pattern statistics rather than deleting them. The key idea is for *CDIA* to during assessment periodically combine the statistics of any access pattern *ap*

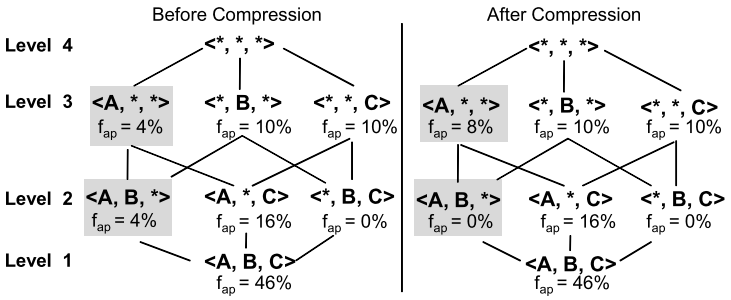

Fig. 7. CDIA example.

whose frequency falls below a preset error rate ϵ with any access pattern node that provides search benefits to *ap* using the DIA lattice. At the end of assessment, *CDIA* returns all access pattern statistics whose frequencies are above preset threshold θ.

Definition 5. Given a state S_t, *DIA* access patterns, threshold θ, error rate ϵ, the set of all possible access patterns of S_t denoted as P_{AP}, the ***CDIA* algorithm** outputs the set of frequency access patterns Q such that: $\forall ap \in Q: F^*_{ap} - \epsilon \leqslant F_{ap} \leqslant F^*_{ap}$ where $(F^*_{ap} = (\forall (k \in P_{AP})((ap \prec k)$ and $(\sum F_k \leqslant \theta))))$.

The *CDIA* approach provides three methods, namely, *insertion* (creating access pattern statistics), *compression* (combining access pattern statistics), and *final results* (finding all access pattern statistics that meet the threshold). The *insert* method is composed of multiple steps. First, the binary representation of the access pattern $B(ap)$ is computed. If an access pattern node exists for *ap*, then its frequency $nd.F_{ap}$ is increased by 1. Otherwise, a new access pattern node is added for *ap* with $nd.F_{ap}$ equal to 1 and maximum error in the frequency δ set to the minimum error rate thus far, or $s_{id} - 1$. After collecting a segment full of search requests, *compression* evaluates the leaf nodes of the DIA lattice. A leaf node is any access pattern node in the DIA lattice that does not provide a search benefit to any other node (i.e., no access pattern node below it has a count > 0). *Compression* proceeds as follows. For each leaf node in the DIA lattice, if the frequency of the access pattern $nd.F_{ap}$ plus the maximum error in the frequency δ is less than the current segment id and a parent of the leaf node exists, then the access pattern count of the leaf node is added to the access pattern count of the parent. Otherwise a new parent node is created with $nd.F_{ap}$ equal to the access pattern count of the leaf node and maximum error in the frequency δ equal to $s_{id} - 1$. Finally, the leaf node is deleted.

During the *final results* step, any access pattern whose frequency is greater than threshold θ is returned as described below. For each node *nd* in the DIA lattice starting from the leaf nodes, first the frequency of the access pattern $nd.F_{ap}$ is computed. If $nd.F_{ap}$ is less than threshold θ, then the count of the access pattern node is added to the parent. Otherwise, the access pattern node is added to the result set Q.

CDIA combination methods. Several strategies can combine the frequency of a child access pattern node nd_c with a parent access pattern node nd_p that provides search benefits to it [23]. One method, *random combination*, randomly picks a parent node. Another method, *highest count combination*, adds the child's frequency to the parent with the highest frequency thus far. The intuition is that the parent node with the largest frequency during assessment has a greater chance of being larger than the preset threshold θ when the final results are found (i.e., at the end of assessment). Our experiments explore both.

CDIA benefits. 1) It guarantees to find access patterns whose frequency is greater than θ. 2) It only stores $\frac{h}{\epsilon} log(\epsilon \sum_{m=0}^{N_{ja}-1} \binom{N_{ja}-1}{m})$ access patterns where h is the number of levels in the DIA lattice [18]. 3) It reduces the number of access patterns stored while retaining the statistics of removed patterns.

Example. Reconsider the above example with a state containing 3 join attributes, a 4 bit *BI*, θ is 5%, and ϵ is .1% (Fig. 7). The CDIA approach using the random combination method combines the frequency of $\langle A1, A2, * \rangle$ into $\langle A1, *, * \rangle$. Selection would find the optimal index configuration.

6. Bounds on statistics reductions

Index selection finds the best *IC* for each state by comparing the dependent costs C_D (Eq. (1)) of each possible *IC* and returns the *IC* with the lowest combined C_D. The "optimal" division of available bits among the states that minimizes the combined C_D must also be considered. The bit division problem given the constraint of minimizing combined C_D across all states has been shown to be equivalent to the multiple-choice knapsack problem (MCKP) [24], a NP-hard problem. Therefore we here consider the problem of locating the optimal *IC* for each state where each state is allocated a specific number of bits. Henceforth *index selection* refers to the problem of finding the *IC* with the lowest combined C_D among all possible *IC*s given B bits.

6.1. Cost calculation reductions

To find the best *IC*, the index configuration dependent cost C_D of every possible combination of *JAS* attributes and B bits must be evaluated. Given N join attributes and B bits, each index configuration with $\leqslant Min(N, B)$ attributes is explored. For each *IC* with bits assigned to k of the N join attributes there are $\binom{B-1}{k-1}$ possible combinations of distributing B bits among the k attributes. Hence the complexity of the search space is: $\sum_{k=1}^{Min(N,B)} \binom{N}{k}\binom{B-1}{k-1}$.

As the size of N or B increases, the search space grows exponentially.

Our AMRI reduces the cost of index selection by eliminating some access frequent statistics. The EPrune algorithm reduces the cost of index selection by eliminating some attributes where there is no benefit to create an index on the attribute. Both use online current statistics rather than compile-time training data to make decisions at runtime.

Before searching over all possible index configurations, the EPrune algorithm [19] applies a benefit function (Eq. (2)) to eliminate any join attributes from the index configuration whose frequency is so low that the overhead for indexing it exceeds the probe cost reduction. If attribute A_x is indexed and assigned N bits, the maximum search cost reduction is $\lambda_r \sum_{ap \in P_A} \lambda_d W_{ap} F_{ap} (1 - \frac{1}{2^N}) C_c$ where P_A denotes the set of access patterns with A_x specified. The additional cost to index attribute A_x is $(\sum_{ap \in P_A} F_{ap}\lambda_r + \lambda_d)C_h$. To determine the maximum benefit of indexing attribute A_x or $MB(A_x)$, we compare the cost to hash attribute A_x to the maximum search cost reduction if attribute A_x is indexed (Eq. (2)). EPrune searches over all possible join attributes and prunes any join attribute A_x where the maximum benefit of indexing A_x is negative.

$$MB(A) = \lambda_r \sum_{ap \in AP} \lambda_d W_{ap} F_{ap} \left(1 - \frac{1}{2^N}\right) C_c - \sum_{ap \in AP} (F_{ap}\lambda_r + \lambda_d) C_h \tag{2}$$

EPrune guarantees to find the optimal *IC* and quickly locates it when the number of join attribute statistics N is small [19]. Complementary to this, our *CSRIA* and *CDIA* methods assist in minimizing the search space by reducing the number of access pattern statistics. This increases the number of possible attributes eliminated by EPrune.

The number of cost calculations to evaluate C_D is driven by the number of access pattern statistics (Section 5.2). A large number of access pattern statistics causes a costly large number of cost calculations for each possible *IC*.

Lemma 1. *Eliminating one access pattern statistic per the* CSRIA *and* CDIA *methods reduces the number of cost calculations performed by* $\sum_{k=1}^{Min(N,B)} \binom{N}{k}\binom{B-1}{k-1}$.

6.2. Statistic reductions

CSRIA and *CDIA* reduce the number of access patterns statistics with the goal of reducing the required memory for assessment and CPU cycles for selection. However, such reductions can influence the quality of the *IC* found.

Lemma 2. *The affect of removing an access pattern on* C_D *is bounded in proportion to the preset threshold* θ *and error rate* ϵ.

Informal proof. An access pattern statistic F_{ap} can have one of the following affects on a given *IC*'s dependent cost C_D (Eq. (1)):

If each attribute in the access pattern *ap* is in the *IC* and vice versa, then all possible bits are used to search (i.e., no wild cards). The portion of C_D that the frequency F_{ap} of *ap* contributes to is $\lambda_r \frac{\lambda_d W_{ap} F_{ap}}{2^B} C_c$. This is the *smallest contribution* that F_{ap} can have on a C_D.

If no attributes in *ap* are in *IC* then a complete scan and comparison of all stored tuples is required. The portion of C_D that F_{ap} contributes to is $\lambda_r \lambda_d W_{ap} F_{ap} C_c$. This is the *largest contribution* that F_{ap} can have on a C_D.

In the worst case removing an access pattern statistic F_{ap} from *CSRIA* removes the largest contribution that a particular F_{ap} can have, or $\lambda_r \lambda_d W_{ap} F_{ap} C_c$, from each possible C_D calculation. Thus any access pattern whose frequency is $\leqslant (\theta - \epsilon)$ is guaranteed to not reduce any C_D by more than $\lambda_r \lambda_d W_{ap} \theta C_c$.

In the worst case removing an access pattern statistic F_{ap} from *CDIA* changes the contribution that F_{ap} has on a C_D from the smallest contribution possible $\lambda_r \frac{\lambda_d W_{ap} F_{ap}}{2^B} C_c$ to the largest contribution possible $\lambda_r \lambda_d W_{ap} F_{ap} C_c$. The potential difference in C_D is $\lambda_r \lambda_d W_{ap} C_c (\frac{F_{ap}}{2^B} - F_{ap})$. Thus any access pattern statistics F_{ap} where the value of $(\frac{F_{ap}}{2^B} - F_{ap})$ is $\leqslant (\theta - \epsilon)$ is guaranteed not to increase any C_D by more than $\lambda_r \lambda_d W_{ap} \theta C_c$. □

7. Scheduling the tuning process

We now introduce four methods that schedule index tuning.

Continuous Index Tuning collects statistics continuously on each state for a fixed period of time. Upon completion of the evaluation window, index selection is executed and assessment is restarted. This method is the simplest to implement and carries no overhead to determine when to initiate index tuning. However when search access patterns do not change, it adds significant assessment overhead even though no new index configuration is required.

Periodic Index Tuning is adapted from index tuning in traditional database systems [25]. At periodic intervals statistics are collected and used to locate the best index configuration. This method is also simple to implement and carries a light overhead to determine when to initiate index tuning. The downfall of this method is when the search access patterns are rapidly changing, it will not identify changes in a state that arise when the state is not being assessed.

Triggered Index Tuning is based upon observed routing changes. It initiates index assessment on states with observed significant routing shifts. In this method, over a fixed period of time the number of search requests processed by each state denoted by λ_r is tracked and compared. Then index assessment is ran on the state with the most significant change. A significant change in λ_r is measured by computing the degree of difference between the current λ_r for a given state S_t and the prior λ_r for S_t. This method is more alert than the periodic index tuning method. It can rapidly identify significant changes in λ_r for a given state. The main drawback of this strategy is that states that never have significant changes in λ_r may never be assessed.

Hybrid Index Tuning employs both periodic and triggered methods. It keeps an ordered list of the states to assessed. When a state has been assessed, the state is moved to the bottom of the assessment list. The advantage of the hybrid method is that it allows the system to catch significant changes in individual states as they occur while ensuring that each state is still periodically assessed.

8. Index migration

When a new "best" index configuration is located, index migration considers whether or not to re-index the current stored tuples to this new index configuration. We now present three alternative migration methods.

8.1. All or nothing

All migration method re-indexes *all* stored tuples to the new "best" index configuration, and removes the old index configuration(s). The result is that each state supports a single common index configuration at all times.

Nothing migration method re-indexes *no* stored tuples (i.e., all stored tuples keep their index configurations). The result of this approach is that states support multiple distinct index configurations. Each tuple locates stored tuples using the time-partitioned indices within their query window (Section 4.2).

Analysis: Is a single (i.e., all migration) better than multiple distinct index configurations (i.e., nothing migration)? To answer this we explore the overhead of migration. Both approaches carry no overhead to evaluate which time-partition indices to migrate. The nothing migration method carries no cost to migrate any stored tuples. While the all migration method carries the cost to migrate all stored tuples. The migration cost of a tuple is the sum of the costs to compute the bucket id of the new index configuration (i.e., new memory location), and to insert the tuple into the new memory location $(C_{insert} + C_{hash,I})$ (Section 5). The deletion cost to remove a tuple is the sum of the costs to compute the memory location of the tuple using the old index configuration and to delete the tuple $(C_{hash,SR} + C_{delete})$. The total migration cost of the all migration method is $\sum_{t \in state}(C_{insert} + C_{hash,I}) + (C_{hash,SR} + C_{delete})$ and the nothing cost is 0.

$$
\begin{aligned}
B_m(IC, BIC) &= C_s(IC) - \big(C_s(BIC) + C_m(IC, BIC)\big) \\
C_s(IC) &= C_{hash,SR} + C_{probe} \\
C_m(IC, BIC) &= \sum_{t \in state} \big((C_{hash,I} + C_{insert}) + (C_{hash,SR} + C_{delete})\big)
\end{aligned} \tag{3}
$$

In order for the migration cost to be worthwhile there must be a benefit in migrating to the new *IC*. Given the index configuration of a state *IC*, the benefit of migrating to the new "best" index configuration *BIC* denoted as $B_m(IC, BIC)$ is the difference between the search costs using *IC* and migration and search cost using *BIC* (Eq. (3)). If $B_m(IC, BIC)$ is greater than 0 then there is a benefit in migrating *IC* to *BIC*.

In order for the *all migration* method to have a lower overall processing cost than *nothing migration* method, the search cost from old inprocess tuples using the new "best" index configuration plus the migration cost must be less than the search cost from old inprocess tuples using their current index configuration, i.e., $(C_s(BIC) + C_m(IC, BIC)) < C_s(IC)$. If this is not the case, it is more cost effective to search for old tuples using their current index configuration and utilize the new "best" index for only new incoming tuples.

8.2. Partial migration

Now the question arises if there is there a benefit to migrate only some *IC*s by precisely determining the benefit of migrating each *IC*. *Partial index migration* supports multiple index configurations, and selectively migrates individual index configurations based upon the estimated future workload of inprocess tuples. We estimate the future workload of each partition by tracking for each partition the number of inprocess tuples by query window (Section 4.2). For each time sliced index configuration in the state, the benefit of migration $B_m(IC, CIC)$ using the estimated number of old inprocess tuples served by the given partition is evaluated. That is, when evaluating $C_s(IC)$ and $C_s(BIC)$ the estimated number of tuples

received within a time unit λ_r is replaced by the number of inprocess tuples within the partition's query window. Only stored tuples from partitions for which there is a benefit in migrating are migrated.

The partial migration method adds the following additional costs to processing: 1) the cost to evaluate each partition whenever a new index configuration is selected, and 2) the cost to track the estimated future workload of old inprocess tuples. Thus in order for the partial migration method to have a lower overall processing cost than nothing migration, the search cost from old inprocess tuples using the new "best" index configuration plus the migration cost and the cost to evaluate the migration needs of each partition must be less than the search cost from old inprocess tuples using the current index configuration(s).

9. Experimental results

9.1. Experimental setup

Metrics. An effective index configuration will reduce tuple response time which ultimately will improve throughput. Thus our experiments compare throughput or the cumulative query results produced over a fixed duration.

Our experimental study explores.

1) Which combination of our proposed index assessment, scheduling, and migration methods is the best at improving the throughput?
2) Is AMRI better than state-of-art approaches at maximizing throughput?
3) How does varying the number of possible query paths affect the throughput of AMRI versus state-of-art approaches?

Synthetic data sets. To test the effectiveness of our methods under conditions where adaptive indices are required, we created synthetic data that deliberately causes intermittent fluctuations in system statistics. In particular, we periodically vary the selectivities of joining each stream to the other streams. More precisely the data content of every 5000 incoming tuples from stream S_i are adjusted so that a different order of STeM operators is the optimal for them. This may cause the router to use new query paths which may lead to index configuration changes.

Real data sets. To demonstrate that AMRI is a viable approach for improving throughput of AMR systems we also test using real data. In particular, we use real data collected from 54 sensors deployed in the Intel Berkeley Research lab between February 28th and April 5th, 2004 (http://berkeley.intel-research.net/labdata/). We partitioned this dataset by sensor in close proximity to each other into several streams. The sensor readings are sent in generation order, as they would have been were the tuples submitted in real-time.

In all approaches, the router periodically sends a few tuples along suboptimal routes to update statistics. The access patterns generated by such tuples have low frequencies. They also have little influence on the final index structure selected. However, as shown in Section 6, they can add additional overhead during index selection. The overhead of these suboptimal searches is included in our experimental results.

Experimental setup/query. All experiments are implemented in CAPE [26] using Linux machines with AMD 2.6GHz Dual Core CPUs and 4GB memory. They run a complete join query across 4 data streams. Each stream joins to each of the 3 other streams via a unique join attribute. Thus each state must efficiently support search requests containing all possible combinations of 3 distinct join attributes, i.e., 7 possible access patterns or $\{A1, *, *\}$, $\{*, A2, *\}$, $\{*, *, A3\}$, $\{A1, A2, *\}$, $\{A1, *, A3\}$, $\{*, A2, A3\}$, $\{A1, A2, A3\}$. There are 6 possible query paths for incoming tuples from stream 1, namely, 1) $S2- > S3- > S4$, 2) $S2- > S4- > S3$, 3) $S3- > S2- > S4$, 4) $S3- > S4- > S2$, 5) $S4- > S3- > S2$, and 6) $S4- > S2- > S3$. Similarly there are also 6 possible query paths for incoming tuples from each of the other streams. In the real data experiments, the query locates sensor data from 4 streams where the temperature, humidity, and relative location (horizontal distance from a fixed point) is the same. Such a query is vital to engineers monitoring building environments.

Our results illustrate that even for systems with a few possible access patterns, there already is a significant benefit in compacting access pattern statistics. Clearly, as the number of possible access patterns increases so does the probability of access pattern statistics being eliminated and thus making more cost savings possible.

The *IC* on each state uses 64 bits. The initial index configurations are found by running index selection using statistics gathered by executing the stream for 15 minutes first to collect quasi-training data. For the state-of-the-art approach, the initial indices are those found to support the most frequent search access patterns by running the training data approach above. Unless otherwise specified, all experiments use the synthetic data set described above.

In each experiment, the partition size is 2500 tuples and the sliding window size is $10,000$ tuples. Each window is composed of 4 partitions. Each partition contributes to any search request whose window contains the stored tuples in the partition (Section 4.2).

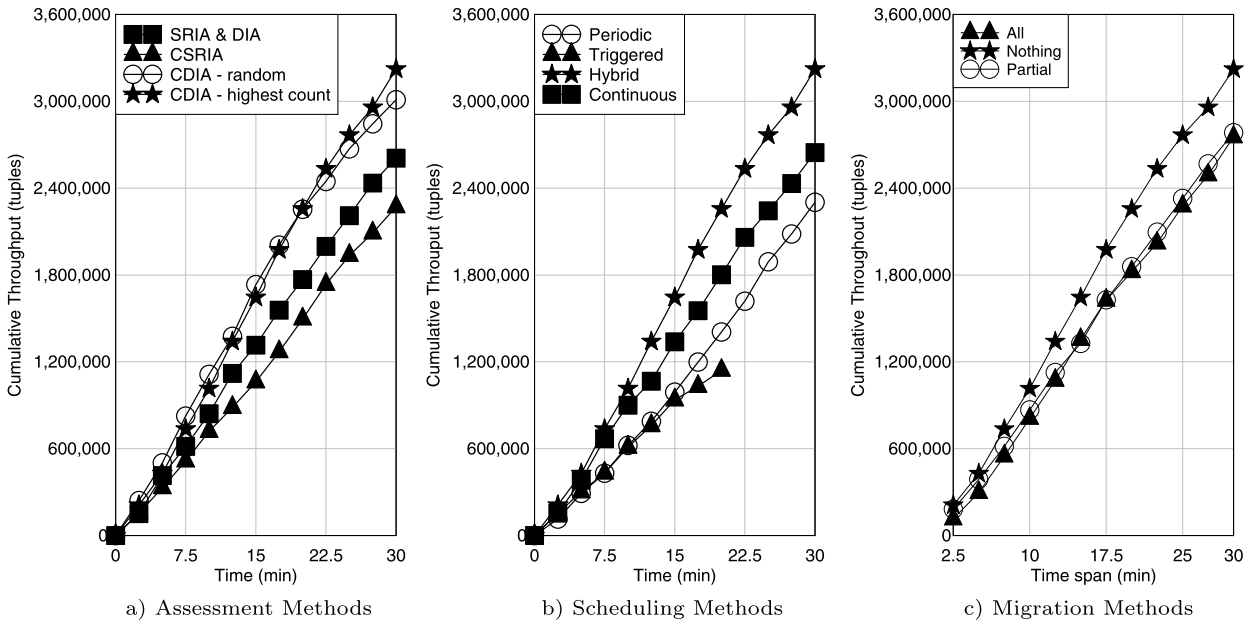

Fig. 8. Alternative tuning methods (maximum error $= \delta = .05$, threshold $= \theta = .1$).

9.2. Evaluating alternative tuning methods

We now explore which of our proposed assessment, scheduling, and migration methods is the best at maximizing throughput.

Index assessment. We first compare the index assessments methods SRIA, CSRIA, DIA, and CDIA using random and highest count compression. Each method is run using the maximum error $\delta = .05$, threshold $\theta = .1$, and hybrid index selection where the periodic and triggered evaluation windows are set to 60,000, and 100,000 search requests respectively. Both CDIA versions (random and highest count compression) outperform DIA, SRIA, and CSRIA (Fig. 8(a)). In fact CDIA using highest count compression outperforms both DIA and SRIA by 19%, and CSRIA by 30%. This demonstrates the utility of combining access pattern statistics (i.e., CDIA) at reducing overhead. The results of DIA and SRIA are rather similar. This is not surprising as they have the same SRIA table format and do not reduce any nodes.

Scheduling method for index tuning. Next, we compare scheduling methods for index tuning namely, *Continuous*, *Periodic*, *Triggered*, and *Hybrid* (Section 6). Each uses the CDIA with highest count compression set up outlined above (i.e., the most effective method). *Hybrid* outperforms *Periodic*, *Triggered*, and *Continuous* by 65%, 29%, and 18% respectively (Fig. 8(b)). This shows the utility of combining traditional periodic scheduling with knowledge gained by the router (i.e., *Hybrid* approach). *Triggered* ran out of memory after 20 minutes. In triggered scheduling, states with no significant changes are never assessed. Thereby such states over time end up having suboptimal index configurations which increases the time to process search requests and delays the processing of incoming search requests.

Index migration. Next, we study the impact of runtime migration methods (Section 8). Each is run using the *Hybrid* scheduling set up outlined above (i.e., the most effective method). *Nothing* migration (i.e., utilizing multiple time-partitioned non-migrating index configurations) outperforms both *All* and *Partial* migration by 19% (Fig. 8(c)). The overhead of migrating the current index configuration to a single new common index configuration (i.e., *All* migration) is greater than the actual search execution time saved. In contrast to traditional systems that support a single common index configuration for the entire state, we conclude that states in AMR systems are best served by multiple time-partitioned non-migrating index configurations. If this were not the case, *All* migration (i.e., utilizing a single common index configuration) would have outperformed *Nothing* migration. Similarly, we also conclude that the overhead to migrate selective index configurations (i.e., *Partial* migration) is greater than the actual search execution time saved.

9.3. State-of-art: number of hash indices

Next, we evaluate the state-of-art AMR indexing approach (i.e., multiple hash indices) [5] by varying the number of hash indices created on each state from the minimum to maximum number of possible indices, 1 to 7 respectively. Static non-adapting hash indices (i.e., selecting an optimal static index configuration at compile-time) produced poor results. Thus

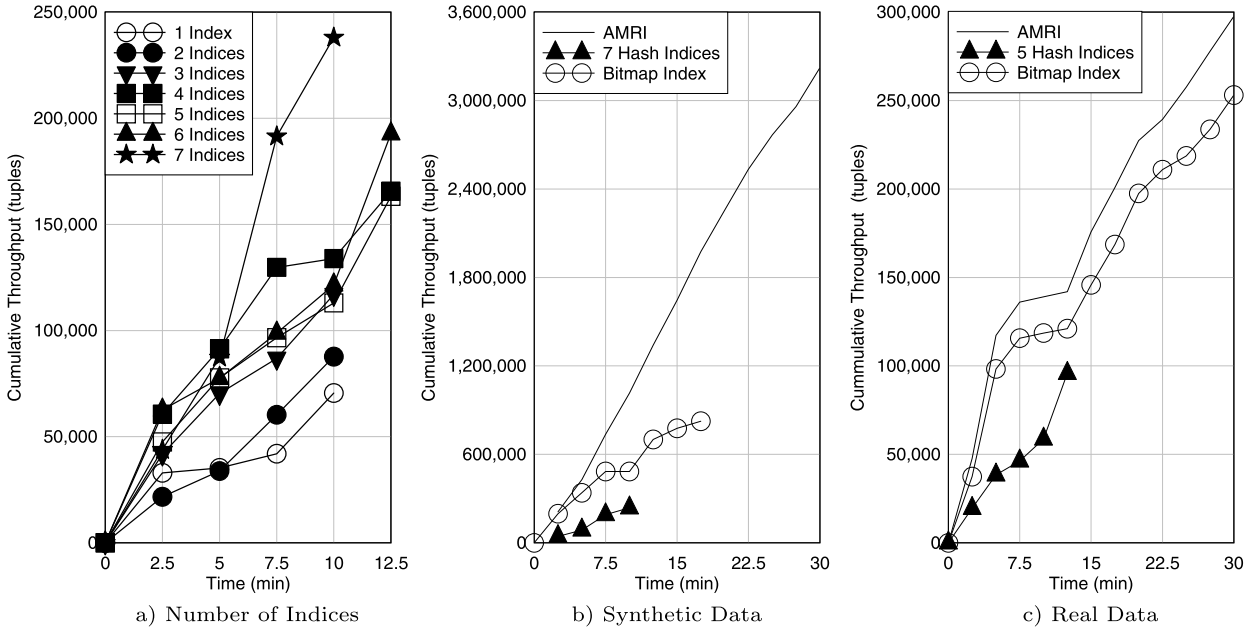

Fig. 9. State-of-art vs AMRI experiments.

instead here we use adaptive hash indices using the nothing migration outlined above and conventional index selection (i.e., indices that support the most frequent access patterns).

All experiments ran for less than 12.5 minutes (Fig. 9(a)). In each experiment, the system ran out of memory due to the CPU and memory overhead (see Section 4.2). For systems with only a few indices a backlog of search requests occurs over time from the processing delay caused by the large number of complete scans performed. The 7 hash indices solution produced more throughput than all others.

9.4. AMRI vs state-of-art

We now explore if AMRI is better than state-of-art approaches [5] at maximizing throughput. First, we compare AMRI to the 7 hash indices solution (i.e., the best state-of-art approach per above) and the state-of-the art static bitmap index. Both AMRI and the static bitmap index start with the same optimal index configuration. A static bitmap index selects an optimal static bitmap index at compile-time. It produced poor results in fluctuating environments such as ours. As we can see, the static bitmap index could not keep up with the search requests and ran out of memory after 15.5 minutes. AMRI is the clear winner. It produces 75% and 93% more results than the non-adapting bitmap index and the best hash index configuration respectively (Fig. 9(b)).

Next, we compare AMRI to the state-of-art approaches [5] using the Intel Berkeley Research lab real data set. AMRI implements CDIA using highest count compression index assessment where the maximum error $\delta = .1$, threshold $\theta = .15$, hybrid scheduling, and nothing migration. We again observed very poor results using non-adapting hash indices. Thus, we also implemented adaptive hash indices as outlined above. AMRI again produces more results than the best hash index solution (68% more) (Fig. 9(c)). Next we compare AMRI to a non-adapting bitmap index. Both start with the same index configuration. In this experiment, AMRI produces 15% more results than the static bitmap index. Compared to the synthetic data, the real data query has a lower selectivity rate. Thus fewer search requests are routed through the system. Hence there are fewer opportunities for any system to gain a substantial lead.

9.5. Skewing the number of possible query paths

We now explore how varying the number of possible query paths affects the throughput of an AMR using AMRI indices versus the state-of-art approach. More precisely, we compare AMRI to adaptive hash indices as defined above when varying the number of possible query paths. Each experiment uses CDIA using highest count compression index assessment where the maximum error $\delta = .1$, threshold $\theta = .15$, hybrid scheduling, and nothing migration (i.e., the best index tuning options). The periodic and triggered evaluation windows are set to $40,000$, and $80,000$ search requests respectively. Synthetic data was formed such that the selectivities of joining one stream to the other streams limits the number of possible query paths favored by the router. More precisely the data value of every 5000 incoming tuples from stream S_i are adjusted so that the number of different optimal orders of STeM operators is controlled.

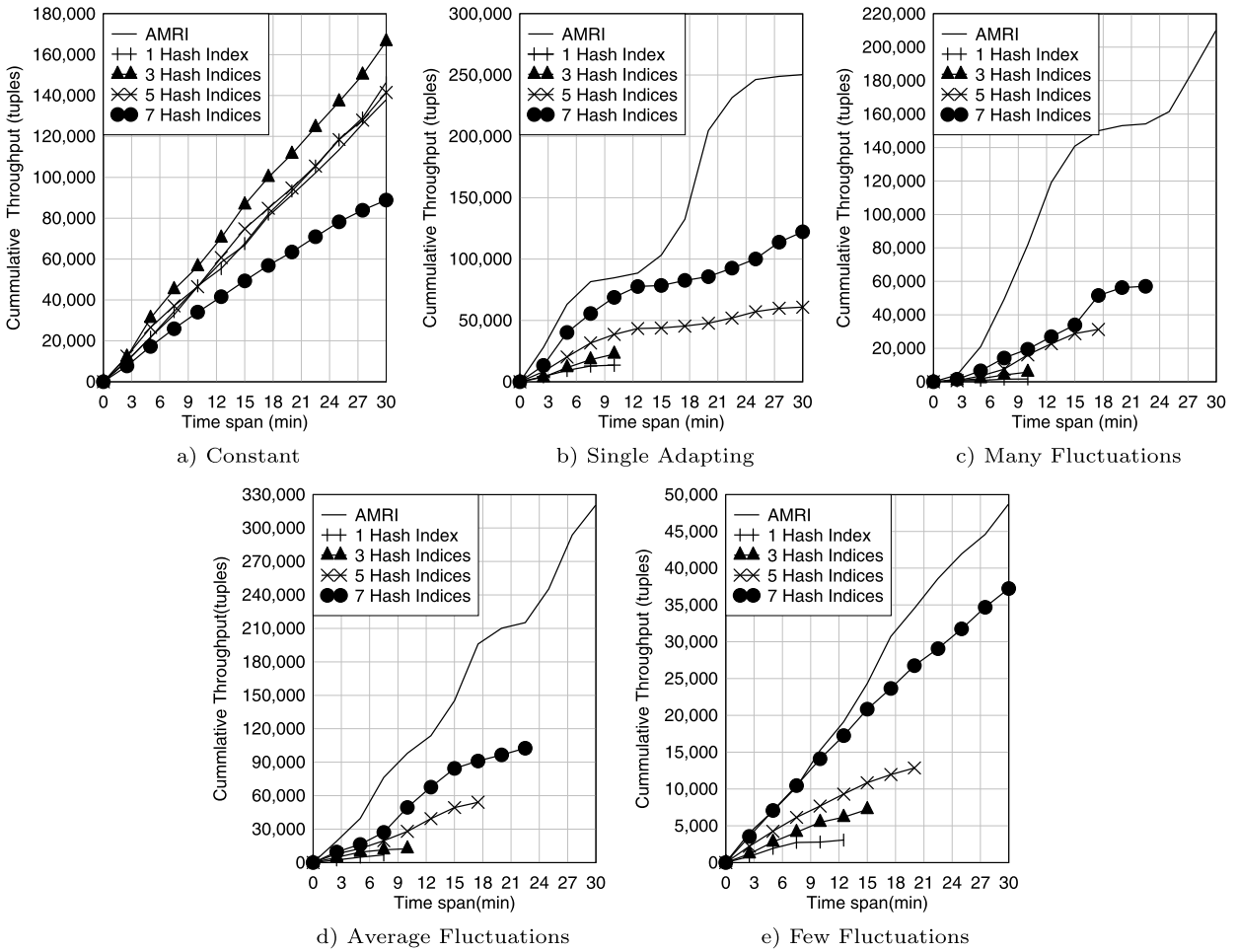

Fig. 10. Skew on the number of query paths experiments.

Constant path. In the *Constant* experiment (Fig. 10(a)), only 1 query path is optimal for the query duration. In other words, 1 non-adapting query path for each stream is selected by the router. AMRI's overhead was not low enough to keep up with the 3 static hash indices solution. But AMRI's overhead was low enough to keep up with both the 1 and 5 static hash indices solutions. However, such a stable non-fluctuating scenario would clearly not be considered a good candidate for an AMR system as the purpose of AMR is to adapt the query plan to fluctuations. Thus we would not deploy our index tuning model in an extremely stable system with known statistics.

Single path. In the *Single Adapting* experiment (Fig. 10(b)) we again consider the case where the router selects a single path for each stream. However, this time, the single path adapts over time. In this case, our AMRI solution produces 35% more results than the highest output produced by the conventional adapting hash indexed based approach. Even for one path, the low overhead of creating and tuning AMRI allows it to produce more results than the state-of-art hash index solutions. This supports our claim that AMRI improves the throughput of queries in fluctuating environments.

Varying the number of paths. Next we vary the number of possible query paths on each stream. Overall AMRI produces more results than the highest output produced by the state-of-art AMR adapting indexing approach. How well AMRI performed is related to the degree of the fluctuations in the number of possible query paths used by the router (Figs. 10(c), 10(d), and 10(e)). A system with *few fluctuations* where 3 streams have 2 possible query paths and 1 stream has 6 possible query paths AMRI produced 24% more results (Fig. 10(e)). A system with *average fluctuations* where 2 streams have 2 possible query paths and 2 streams have 6 possible query paths AMRI produced 43% more results (Fig. 10(d)). A system with *many fluctuations* where 1 stream has 2 possible query paths and 3 streams have 6 possible query paths AMRI produced 73% more results (Fig. 10(c)). Clearly in these scenarios, AMRI adapts to systems with higher fluctuations better than the state-of-art approach.

9.6. Summary of results

We found that:

1) of our proposed index assessment methods, CDIA using highest count compression was better at improving throughput than DIA, SRIA, and CSRIA;
2) of our proposed scheduling methods, Hybrid was better at improving throughput than Continuous, Periodic, and Triggered;
3) of our proposed migration methods, None was better at improving throughput than All and Partial. In other words, given the temporary nature of index usage in the streaming context, rebuilding indices is generally not beneficial;
4) our AMRI, using both synthetic and real data, was better at improving throughput than the best state-of-art and non-adapting bitmap index solution;
5) our AMRI adapted to systems with heavy route fluctuations better than the best state-of-art solution.

10. Related work

While traditional database systems focus on processing queries over persistent storage, a new area has emerged focusing on continuous query processing over data streams [27–36]. Since stream data is infinite, new query semantics have been defined to avoid infinite data pile-up and thus unpredictable latency. Window-semantics for query operators are introduced to remove expired data from the database state [37], and with it new algorithms [38–41]. Dynamic constraints are also interleaved in the actual data stream [42,43]. To address system dynamicity, plan optimization and migration for static databases such as [44,45] are no longer sufficient. Instead online query plan optimization and run-time migration become critical techniques for maximal adaptivity [33,37].

Most major commercial DBMSs [46–48] determine a query plan at compile-time. This has been shown to lead to optimization coarseness, or not efficiently processing all tuples [49,50]. To combat this, AMR systems at runtime decide for each incoming tuple which operator to visit next, e.g., Eddy [3,4,7,6], *QMesh* [51], and CBR [52]. The first AMR system, Eddy, used a router to route individual tuples through operators [3]. Enhancements in routing policies were proposed in [4,7]. Multi-query AMR issues were covered by [6]. [53] adds batching to Eddies to reduce the tuple-level routing overhead. *QMesh* [51] groups and routes tuples along preselected operator orderings based on data stream characteristics. *Content-based routing* (CBR), an extension of Eddies [52], focuses on continuously profiling operators and identifying attributes to partition and route incoming tuples by.

We borrow the concept of the SteM (a State Module) from [5]. Stem is used with an Eddy to allow flexible adaptation of join algorithms, access methods, and the join spanning tree. The STAIR operator [54] encapsulates the state typically stored inside the join operators and exposes it to the Eddy.

Index Tuning in static databases aims to find a set of indexes that maximally benefit a query workload by either selecting the optimal index configuration off-line [8,9] or online during execution [12–14] (i.e., [12–14] in our original submission). [12] explores performance index tuning methods that exploit the ordering of statements in the workload of traditional databases. In particular, they consider the sequence of insertion and update requests as compared to search requests. We instead seek performance tuning methods for adaptive multi-route streaming databases where we exploit the sequence of operators that an incoming tuple visits. [13], a demonstration paper, introduced COLT (Continuous On-Line Tuning), a self-tuning framework that adjusts the quality and quantity of statistics gathered. That is, it reduces the overhead to gather statistics when the system is performing well, and increases the overhead when it is not. However, unlike our work, they do not have a bound on the effect that decreasing or increasing the statistics collected has on the quality of the index configuration found. In other words, their method is simplistic. Our AMRI approach is more sophisticated in that it reduces the overhead while maintaining the integrity of the index selected (i.e., bounds on the optimality (i.e., quality) of the index configuration found) within a preset threshold and error rate. [14] introduced an online algorithm that continuously monitors changes in the workload and data to tune indices in traditional databases. One of the main issues they address is that given the data is predominately static how to ensure that the single index configuration used does not keep being switched back and forth. Our problem is very different. Namely, in streaming databases the data will only be stored for a short duration of time due to the moving time windows over the streams. Thus, the data will only be used to create results for a subset of the incoming stream that are within the proper query window. Hence, we seek to identify the best index configuration for subsets of continuous tuples in the incoming streams. That is, we seek to support multiple time-spliced index configurations that live for short durations.

Indexing in stream contexts has not yet received much attention, possibly due to the dynamic nature of DSMS. [55] studies methods for indexing a single attribute for stream algebra operators under sliding window-semantics. [19] proposed the EPrune algorithm for DSMS where the search request workload is known prior to execution. The EPrune algorithm [19] eliminates any join attributes from being considered during index selection whose frequency is so low that the overhead for indexing it exceeds the probe cost reduction. We instead tackle the opposite and harder problem, namely, when the search requests continuously fluctuate and are not known at compile-time. In addition, we explore novel approaches that cover the index tuning life cycle for AMRs.

Bit-address indexing, initially designed to index partial match queries on a file database [56], has been applied on applications ranging from compactly storing very large multidimensional arrays [57] to reducing processing costs of multi-dimensional queries [58,59]. [19] studied index selection heuristics using a single bit-address index where the search request workload is known prior to execution. We instead tackle the on-line index tuning problem for AMRs.

Assessment methods for the bit-address index design have not been studied while they are core to our effort. Such methods utilize stream sampling algorithms. Our work is the first to put forth for the first time the insight that heavy hitter algorithms [60,61,17], a type of stream sampling, uniquely meets the requirements of AMRs (Section 1).

[60] introduced the first deterministic algorithm for approximating frequency counts, a.k.a. the heavy hitter method. [17] added the error rate ϵ approximation guarantee. Our *CSRIA* method is modeled after [17]. However, we do not use raw data such as [17], but instead focus on the modeling and measuring search requests and their frequencies. Hierarchical heavy hitter, applying the heavy hitter methodology to hierarchical multidimensional data, was studied in [18,62] to solve network traffic problems. Our *CDIA* method is modeled after this hierarchical heavy hitter work. CDIA implements two compression methods, namely, *random combination* and *highest count combination* which are variations of compression methods presented by [23]. In our context, we customize the compression methods to handle the search benefit relationships between indices in AMRs. To our knowledge, ours is the first application of heavy hitter and hierarchical heavy hitter algorithms to address the problem of index tuning in AMRs.

11. Conclusions

AMRs require an index design and on-line tuning that is light weight with respect to both CPU and memory resources and yet efficiently can support diverse and ever adapting search criteria caused by the adapting query plans in AMRs. It is challenging to identify query path fluctuations, and to quickly and precisely adjust index configuration(s) to best serve the new query path(s). To address this problem we developed the Adaptive Multi-Route Index for AMR systems or AMRI. AMRI employs multiple time-partitioned bitmap indices to serve a workload of diverse access patterns with partially overlapping query windows. We introduced Compact Dependent Index Assessment which improves query responsiveness by reducing the selection search time. Hybrid Index Selection implements both standard periodic index selection and triggering index selection based upon changes in routing. We established the optimality bounds on the quality of the index configuration found during selection. We also are the first to demonstrate that a state utilizing multiple time-spliced index configurations significantly improved the throughput of AMR systems.

Our experiments validated that AMRI improves the throughput in dynamic stream environments while keeping the overhead minimal. In particular, using synthetic data AMRI produced on average 93% more results than the state-of-art approach. With our real data set AMRI produced on average 68% more results. We also evaluated multiple index scheduling and migration methods and determined their relative effectiveness.

Acknowledgments

We thank Luping Ding and our WPI peers for feedback, and GAANN and NSF grants: IIS-1018443, 0917017, 0414567, and 0551584 for financial support.

References

[1] D.J. Abadi, et al., Aurora: a new model and architecture for data stream management, in: VLDB, 2003, pp. 120–139.
[2] T. Urhan, et al., Cost-based query scrambling for initial delays, in: SIGMOD, 1998, pp. 130–141.
[3] R. Avnur, et al., Eddies: continuously adaptive query processing, in: SIGMOD, 2000, pp. 261–272.
[4] P. Bizarro, et al., Content-based routing: different plans for different data, in: VLDB, 2005, pp. 757–768.
[5] V. Raman, et al., Using state modules for adaptive query processing, in: ICDE, 2003, pp. 353–364.
[6] S. Madden, et al., Continuously adaptive continuous queries over streams, in: SIGMOD, 2002, pp. 49–60.
[7] F. Tian, et al., Tuple routing strategies for distributed eddies, in: VLDB, 2003, pp. 333–344.
[8] S. Agrawal, et al., Automated selection of materialized views and indexes in SQL databases, in: VLDB, 2000, pp. 496–505.
[9] S. Chaudhuri, et al., An efficient cost-driven index selection tool for microsoft SQL server, in: VLDB, 1997, pp. 146–155.
[10] S. Agrawal, et al., Database tuning advisor for microsoft SQL server, in: SIGMOD, 2005, pp. 930–932.
[11] B. Dageville, et al., Automatic SQL tuning in oracle 10g, in: VLDB, 2004, pp. 1098–1109.
[12] S. Agrawal, et al., Automatic physical design tuning: workload as a sequence, in: SIGMOD, 2006, pp. 683–694.
[13] K. Schnaitter, et al., Colt: continuous on-line tuning, in: SIGMOD, 2006, pp. 793–795.
[14] N. Bruno, S. Chaudhuri, An online approach to physical design tuning, in: ICDE, 2007, pp. 826–835.
[15] K. Schnaitter, et al., On-line index selection for shifting workloads, in: ICDE, 2007, pp. 459–468.
[16] K. Works, et al., Index tuning for adaptive multi-route data stream systems, in: Scalable Stream Processing Systems, 2010, pp. 1–8.
[17] G.S. Manku, R. Motwani, Approximate frequency counts over data streams, in: VLDB, 2002, pp. 346–357.
[18] G. Cormode, et al., Finding hierarchical heavy hitters in data streams, in: VLDB, 2003, pp. 464–475.
[19] L. Ding, E.A. Rundensteiner, Index tuning for parameterized streaming groupby queries, in: Scalable Stream Processing Systems 2008, pp. 68–78.
[20] A. Deshpande, J.M. Hellerstein, Lifting the burden of history from adaptive query processing, in: VLDB, 2004, pp. 948–959.
[21] J. Kang, J.F. Naughton, S.D. Viglas, Evaluating window joins over unbounded streams, in: ICDE, 2003, pp. 341–352.
[22] T. Johnson, S. Muthukrishnan, I. Rozenbaum, Sampling algorithms in a stream operator, in: SIGMOD, 2005, pp. 1–12.
[23] G. Cormode, et al., Diamond in the rough: finding hierarchical heavy hitters in multi-dimensional data, in: SIGMOD, 2004, pp. 155–166.
[24] A. Mohr, Bit allocation in sub-linear time and the multiple-choice knapsack problem, in: Data Compression Conference, 2002, pp. 352–361.

[25] M. Hammer, A. Chan, Index selection in a self-adaptive data base management system, in: SIGMOD, 1976, pp. 1–8.
[26] E.A. Rundensteiner, et al., CAPE: continuous query engine with heterogeneous-grained adaptivity, in: VLDB, 2004, pp. 1353–1356.
[27] R. Motwani, et al., Query processing, resource management, and approximation in a data stream management system, in: CIDR, 2003, pp. 245–256.
[28] J.M. Hellerstein, et al., Adaptive query processing: technology in evolution, IEEE Data Eng. Bull. 23 (2) (2000) 7–18.
[29] A. Gounaris, N.W. Paton, A.A. Fernandes, R. Sakellariou, Adaptive query processing: a survey, in: BNCD, 2002.
[30] A. Arasu, et al., Stream: the stanford stream data manager, in: SIGMOD Demo, 2003.
[31] D. Abadi, et al., Aurora: a data stream management system, in: SIGMOD Demo, 2003, p. 666.
[32] S. Chandrasekaran, et al., TelegraphCQ: continuous dataflow processing, in: SIGMOD Demo, 2003, p. 668.
[33] Z. Ives, et al., An adaptive query execution system for data integration, in: SIGMOD, 1999, pp. 299–310.
[34] J. Chen, D. DeWitt, F. Tian, Y. Wang, NiagaraCQ: a scalable continuous query system for internet databases, in: SIGMOD, 2002, pp. 379–390.
[35] E.A. Rundensteiner, et al., Cape: continuous query engine with heterogeneous-grained adaptivity, in: VLDB Demo, 2004, pp. 1353–1356.
[36] T. Sutherland, B. Pielech, Y. Zhu, L. Ding, E.A. Rundensteiner, An adaptive multi-objective scheduling selection framework for continuous query processing, in: IDEAS, 2005, pp. 445–454.
[37] D. Abadi, et al., Aurora: a new model and architecture for data stream management, in: VLDB, 2003, pp. 120–139.
[38] J. Kang, J.F. Naughton, S.D. Viglas, Evaluating window joins over unbounded streams, in: ICDE, 2003, pp. 341–352.
[39] L. Golab, M.T. Ozsu, Processing sliding window multi-joins in continuous queries over data streams, in: VLDB, 2003, pp. 500–511.
[40] M.A. Hammad, M.J. Franklin, W.G. Aref, A.K. Elmagarmid, Scheduling for shared window joins over data streams, in: VLDB, 2003, pp. 297–308.
[41] A. Arasu, J. Widom, Resource sharing in continuous sliding-window aggregates, in: VLDB, 2004, pp. 336–347.
[42] P.A. Tucker, D. Maier, T. Sheard, L. Fegaras, Exploiting punctuation semantics in continuous data streams, TKDE 15 (3) (2003) 555–568.
[43] L. Ding, E.A. Rundensteiner, Evaluating window joins over punctuated streams, in: CIKM, 2004, pp. 92–101.
[44] N. Kabra, D. Dewitt, Efficient mid-query re-optimization of sub-optimal query execution plans, in: SIGMOD, 1998.
[45] G. Graefe, R. Cole, Optimization of dynamic query evaluation plans, in: SIGMOD, 1994, pp. 150–160.
[46] M.S. Server, http://www.microsoft.com/sql/default.mspx.
[47] DB2, http://www.ibm.com/software/data/db2/.
[48] Oracle, http://www.oracle.com/index.html.
[49] G. Graefe, W.J. McKenna, The volcano optimizer generator: extensibility and efficient search, in: ICDE, 1993, pp. 209–218.
[50] M.M. Astrahan, et al., System r: relational approach to database management, TODS 1 (2) (1976) 97–137, http://dx.doi.org/10.1145/320455.320457.
[51] R.V. Nehme, et al., Self-tuning query mesh for adaptive multi-route query processing, in: EDBT, 2009, pp. 803–814.
[52] P. Bizarro, et al., Content-based routing: different plans for different data, in: VLDB, 2005, pp. 757–768.
[53] A. Deshpande, An initial study of overheads of eddies, SIGMOD Rec. 33 (1) (2004) 44–49.
[54] A. Deshpande, et al., Lifting the burden of history from adaptive query processing, in: VLDB, 2004, pp. 948–959.
[55] L. Golab, S. Garg, M.T. Ozsu, On indexing sliding windows over on-line data streams, in: EDBT, 2004, pp. 712–729.
[56] A.V. Aho, J.D. Ullman, Optimal partial-match retrieval when fields are independently specified, ACM Trans. Database Syst. (1979) 168–179.
[57] E.J. Otoo, et al., Chunking of large multidimensional arrays, Tech. Rep. LBNL-63230, Lawrence Berkeley National Laboratory, February 2007.
[58] D. Rotem, K. Stockinger, K. Wu, Towards optimal multi-dimensional query processing with bitmapindices, Tech. Rep. LBNL–58755, Lawrence Berkeley National Laboratory, September 2005.
[59] D. Rotem, et al., Minimizing i/o costs of multi-dimensional queries with bitmap indices, in: Scientific and Statistical Database Management, 2006, pp. 33–44.
[60] J. Misra, D. Gries, Finding repeated elements, Tech. Rep., Cornell University, Ithaca, NY, USA, 1982.
[61] R.M. Karp, et al., A simple algorithm for finding frequent elements in streams and bags, ACM Trans. Database Syst. (2003) 51–55.
[62] C. Estan, G. Varghese, New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice, ACM Trans. Comput. Syst. (2003) 270–313.