WPI-CS-TR-10-05 February 2010

Time-partitioned Index Design for Adaptive Multi-Route Data Stream Systems utilizing Heavy Hitter Algorithms

by

Karen Works
Elke Rundensteiner
and Emmanuel Agu

Time-partitioned Index Design for Adaptive Multi-Route Data Stream Systems utilizing Heavy Hitter Algorithms

Karen Works, Elke A. Rundensteiner, and Emmanuel Agu
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609
Tel.: (508) 831–5857, Fax: (508) 831–5776
{kworks, rundenst,emmanuel}@cs.wpi.edu

Abstract

Adaptive multi-route query processing (AMR) is a recently emerging paradigm for processing stream queries in highly fluctuating environments. AMR dynamically routes batches of tuples to operators in the query network based on routing criteria and up-to-date system statistics. In the context of AMR systems, indexing, a core technology for efficient stream processing, has received little attention. Indexing in AMR systems is demanding as indices must adapt to serve continuously evolving query paths while maintaining index content under high volumes of data. Our proposed Adaptive Multi-Route Index (AMRI) employs a bitmap time-partitioned design that while being versatile in serving a diverse ever changing workload of multiple query access patterns remains lightweight in terms of maintenance and storage requirements. In addition, our AMRI index design and migration strategies seeks to met the indexing needs of both older partially serviced and newer incoming search requests. We show that the effect on the quality of the index configuration selected based on using AMRIs compressed statistics can be bounded to a preset constant. Our experimental study using both synthetic and real data streams has demonstrated that our AMRI strategy strikes a balance between supporting effective query processing in dynamic stream environments while keeping the index maintenance and tuning costs to a minimum. Using a data set collected by environmental sensors placed in the Intel Berkeley Research lab, our AMRI outperforms the state-of-the-art approach on average by 68% *in cumulative throughput.*

1 Introduction

1.1 Index Tuning

As predicted by Abadi et. al. [1] the number of monitoring applications has soared. In addition, many monitoring applications that were once simple (i.e., supported by a single simple query) have become complex (i.e., supported by multiple complex queries that share data). Consider the stock market. A few years ago, an analyst looking to trade stock at the best price may have only considered the current price and volume of that stock. Today, to stay ahead of rapid market shifts, the same analyst needs to combine current price and volume data with the latest information on the company as well as that sector of the market. Such company and sector information is now available from multiple sources (i.e., news feeds, web sites, and blogs). In short, monitoring applications are increasingly requiring complex data stream queries.

Modern data stream management systems (DSMS) that support these complex monitoring applications must efficiently function over long periods of time in environments that are susceptible to frequent fluctuations in data arrival rates [32]. Such fluctuations cause periodic variances in the selectivity and the performance of operators, typically rendering the most carefully chosen query plan sub-optimal and possibly ineffective [6]. This has driven research into DSMS that continuously adapt the best query path for sets of tuples, henceforth referred to as Adaptive Multi-Route query processing systems (AMR) [6, 7, 24, 20, 31]. AMR systems allow the order in which operators are executed (i.e., the query path) to adapt to current system statistics. The prominent AMR system, Eddy [6], utilizes a central routing operator that decides for sets of tuples which operator to visit next based upon the system environment.

AMR systems selectively adapt query paths for incoming tuples based upon characteristics of the tuples (i.e., a distinct order in which operators are visited may be chosen for different tuples). For that reason, join operators in AMRs are required to process search requests generated from intermediate results where the incoming intermediate results may have been routed along rather different query paths before they arrive at the join operator. Consider a query composed of multiple join operators. In such a query, the query path taken by a tuple determines which tuples are in the intermediate result. Values in the tuples that embody an intermediate result are used to generate a search request for locating tuples in a given state that match the intermediate tuple. Consider two tuples t_1 and t_2 from $StreamA$. t_1 is first routed to join with tuples from $StreamB$ and then to join with tuples from $StreamC$. While t_2 is first routed to the join with tuples from $StreamC$. To efficiently join with tuples from $StreamC$ requires the system to be able to locate tuples using search criteria that include the join attributes between $StreamA$ and $StreamC$ (i.e., tuple t_2) as well as the combined join attributes of $StreamA$ and $StreamB$ and their relation to $StreamC$ (i.e., tuple t_1). Current monitoring systems cannot afford to create indices for every type of search request (i.e., for all possible query paths) given the complexity of the number of queries and joins. Therefore when fluctuations occur and the query paths change in a significant manner, it is important for AMRs to quickly and accurately adjust the index configurations to best serve the new set of query path(s). However this challenging indexing problem has been ignored in the literature until now. This is now the focus of our work.

1.2 Indexing in AMR Systems

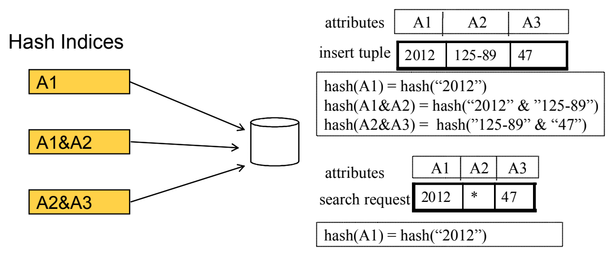
Little work has been done on indexing in this dynamic adaptive multi-route context. The main work that we are aware of in this area is [24]. Raman et. al. [24] proposed to create multiple access modules for each state where a state stores tuples originating from a single stream (i.e., similar to tables in traditional databases). Each access module employs a hash index on a subset of attributes to optimize a particular data access on such a state. We now illustrate the inefficiencies of such a system.

Example: Consider a DSMS which tracks the current location of packages using a network of sensors. Each sensor propagates 3 attributes: *priority code* ($A1$), *package id* ($A2$), and *location Id* ($A3$) to the central application. The state storing the sensor data has 3 hash indices which support attribute combinations $A1$, $A1\&A2$, and $A2\&A3$, respectively (Figure 1). To insert tuple t, first t is stored into the state. Then hash keys are created for $t.A1$, $t.A1\&t.A3$, and $t.A2\&t.A3$, each is linked to t, and stored in the respective index.

Consider the search request sr_1 looking for all packages with priority code $A1$ = 2012 and location id $A3$ = '47'. Executing sr_1 involves first determining the most suitable hash index for processing sr_1. The most suitable hash index is the hash index that supports the largest number of attributes in sr_1 and contains no attributes not in sr_1. In this case, it is hash index $A1$. Then a lookup is performed via $A1$.

Now consider sr_2 looking for all packages where location id $A3$ = '47'. The only attribute referenced in sr_2 is $A3$. Since no suitable hash index exists for sr_2, a full scan of the state must be performed. To improve the search time of sr_2 requires the creation of a new hash index on $A3$ only. But this would henceforth add additional

Figure 1. Indexing in AMR Systems



memory and maintenance costs for each tuple stored in the state.

In short, to support such diversity of criteria requires a possible large number of hash indices. This not only adds a high memory overhead due to multiple references required for each stored tuple, but worse yet considerable maintenance costs to support the multiple hash indices under the heavy update loads experienced by DSMS. The inefficiency of deploying multiple hash indices over a single state is confirmed by our experimental study on both synthetic and real data sets (See Section 8).

1.3 AMR Index Requirements

Traditional off-line index tuning approaches that select the "best" fixed index structure off-line [3, 9, 2, 12] are not adequate for AMR systems where the search request workload is in constant flux. On-line index tuning approaches continuously evaluate and adjust the index structure during execution via the index tuning life cycle [4, 29, 8, 30]. The steps in the life cycle are: 1) to gauge changes in the search request workload (*assessment*), 2) to find the "best" index structure (*selection*), and 3) to re-index states using their current "best" index structure (*migration*). Clearly, AMR systems require an online index tuning solution that distinctly addresses the particular challenges of AMR systems.

AMR systems have unique challenges. 1) Periodically the router sends search requests to suboptimal operators to update system statistics [6]. These suboptimal access patterns have extremely low frequencies. Although these suboptimal access patterns are not likely to influence the final index structure selected, they add additional overhead to both index assessment and selection. To lower such overhead requires AMR systems to adjust the quality and quantity of statistics collected. If too few statistics are kept the overhead may be reduced but at the expense of not being able to locate the most optimal index configuration. Whereas if too detailed statistics are kept, the optimal index may be located but it may require a higher overhead. 2) AMR systems must continuously assess and adjust indices to best support the query paths used by search requests in the system. The abruptness and frequency of changes in the query paths in AMR systems make this extremely challenging. Furthermore the overhead of assessing indices clearly must not detract from producing rapid results. 3) The router continuously evolves query paths used to process new search requests. As the query paths evolve, the system will contain older partially serviced search requests processed along query paths different from the newly established "best" query paths for new search requests. To minimize the query response time requires efficient processing of both older partially serviced and newly incoming search requests within the same system.

In short, AMR systems require an index design and on-line tuning solution that meets the conflicting demands of being light weight with respect to both CPU and memory and yet efficiently supporting multiple possibly rather diverse criteria for an ever adapting workload of search requests. Such a solution would enhance any AMR system and thus complex monitoring applications making use of an AMR system by reducing processing time while minimizing the overhead required.

1.4 The Proposed Approach: AMRI

AMR systems require an index design that: 1) supports a large number of rather diverse criteria, 2) requires minimal maintenance time, 3) is compact in size so to exist in main memory, 4) is able to efficiently process all search requests regardless of their varying criteria and proximity to completion.

An AMR index tuning systems must : 1) efficiently and accurately identify key and abrupt changes in the query paths while not detract from producing rapid results (*assessment*), 2) maintain the quality of the best index configuration selected within a preset threshold while using compact statistics (*selection*), and 3) efficiently determine the best approach to re-indexing tuples already stored in a state to support efficient processing of both older partially serviced and newly incoming search requests (*migration*). Our proposed *Adaptive Multi-Route Index* (*AMRI*) solution incorporates a physical index design and customized index tuning methods that meet the above named requirements of AMR systems.

Our earlier work on index tuning has been published in the proceedings of the SSPS [17]. In this earlier work, we focussed on the design of two index assessment methods called *Compact Self Reliant Index Assessment CSRIA*, and *Compact Dependent Index Assessment CDIA* that reduce the system resources required by index assessment by eliminating statistics while maintaining the integrity of the index configuration. *CSRIA* utilizes a heavy hitter method [21] to track and compact statistics. *CDIA* tracks statistics in a lattice, exploiting the dependent relationships between access patterns, and employs a hierarchical heavy hitter method [10] to compact the statistics.

In this paper, we extend our basic index solution to efficiently process all search requests regardless of their varying criteria or proximity to completion as we now seek to support the diverse criteria of both older partially serviced and new incoming search requests. We also now establish the bounds on the optimality (i.e., quality) of the index configuration found during index selection using the set of statistics generated by our *CSRIA* and *CDIA* index assessment methods (i.e., compact statistics). We further extend our index tuning strategies to incorporate scheduling assessment methods (i.e., index selection scheduling) as well as efficient index migration methods. Our new contributions include:
1) We enhance our index design to support all search requests regardless of their varying criteria or proximity to completion by employing a bitmap time-partitioned structure.
2) We formulate and then prove that the effect of statistic compaction on the potential loss of quality of the selected index configuration can be bounded by a preset constant.
3) We propose customized methods for scheduling index tuning. In particular, a *Triggered Index Tuning* method to trigger index assessment based on observed routing changes. This allows the system to nimbly adapt to relevant changes by scheduling assessment on states with observed significant routing shifts.
4) We propose a *Partial Index Migration* that migrates time slices of an index based on their estimated future use. Through our study of index migration approaches we explore the question of whether utilizing a single or multiple index configurations is best at improving the throughput of all search requests in the system regardless of their proximity to completion.
5) We conduct experiments on evaluating the effectiveness of proposed scheduling assessment methods and index migration methods using both real and synthetic data. We demonstrate that our *AMRI* solution always wins over the current AMR indexing methods, including traditional hash indices [24] and bitmap indexing [13]. We also explore the effect the selectivity of query operators has on the performance of our AMRI solution.

The paper is organized as follows. Section 2 defines terminology. Section 3 describes the physical index design of AMRI, and Section 4 presents the proposed index assessment methods. Section 5 bounds the effect of compacting statistics on the index configuration selected. Section 6 presents our index tuning scheduling methods, while Section 7 presents our index migration methods. Sections 8, 9, and 10 cover experimental analysis, related work, and conclusions, respectively.

2 Preliminary

We consider SPJ (select-project-join) queries processed under a *suffix* window (Figure 2). A suffix window indicates the length of the data history to be queried using standard sliding window semantics. The solution is explained using a single SPJ query. However, our proposed logic equally applies to AMR systems running multiple SPJ queries.

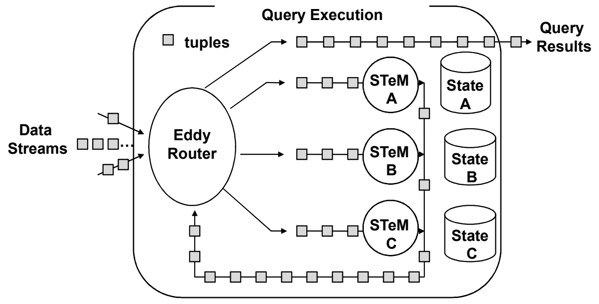
Figure 2. SPJ Query Template and Example

```
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
```

A *state* is instantiated for each stream in the FROM clause. In our example, a state is created for each of the following streams: StreamA, StreamB, and StreamC. Each state is associated with a unary join *STeM operator* [24] that supports insertion and deletion of tuples, and locating of tuples based on join predicates. Figure 3 depicts the AMR of the query outlined in Figure 2.

Figure 3. AMR of Example



A *join predicate* is expressed in the WHERE clause of a query composed of 1) an attribute stream reference, 2) a join expression ($=$, $<$, $>$ $\geq$, $\leq$), and 3) another attribute stream reference (e.g., $A.A1 = B.A2$). The *join attribute set JAS* for a state is the set of all attributes specified in at least one join predicate in the query (i.e., for Stream A JAS = $\{$A1,A3$\}$). Each STeM operator can search for tuples in its state based on 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 a combination of attributes of a *JAS* used to specify a search. An ap is denoted by a vector whose size is equal to the number of attributes in *JAS*. Each vector position represents a single join attribute. A join attribute used to search is represented by the capital letter naming the attribute, while a join attribute not used is represented by the wild card symbol $*$.

Reconsider the state of Stream A where JAS = $\{$A1,A3$\}$. $< A1, A3 >$ is an ap indicating that all join attributes are used to search. While $< A1, * >$ is an ap only using join attribute $A1$, and $< *, * >$ is an ap using no join attributes (i.e., a full scan).

3 Adaptive Multi-Route Index

3.1 Physical Index Design

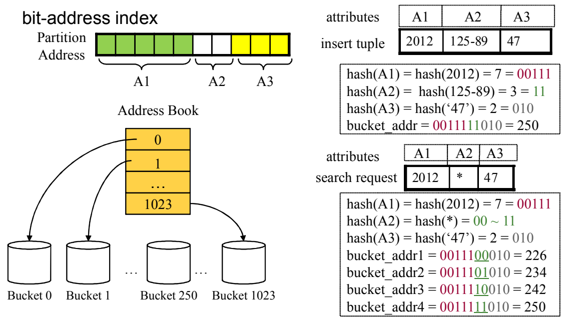
Instead of maintaining multiple distinct hash indices we propose to employ a versatile yet compact bit-address index as the foundation of our solution [13]. We now explain how this meets the AMR requirements covered in Section 1.4.

The foundation of our bit-address index is an index key map (also called the *index configuration* IC) that is a blueprint to the memory location where tuples are stored. Given B bits, 2^B bucket locations are used to store tuples. The specific bucket where a tuple is stored is found by using the IC to map a tuple's attribute values to a bucket location. IC delineates for each join attribute the number of bits (possibly none) used in mapping. No index links are ever created as the IC derived for each tuple is never stored. This significantly reduces the amount of memory and CPU time required.

Reconsider the example from Section 1.2. Now we insert tuple t, a *package* record, into the bit-address index BI solution in Figure 4 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 is generated by mapping the values for $t.A1$, $t.A2$, and $t.A3$, which are 00111, 11, and 010 respectively. Then these values are combined into form the bucket id. 0011111010 represents bucket 250. Thus t is stored in bucket 250. t does not store the IC generated. Thus, no memory or CPU time is required to support index links. This is in contrast to the hash index approach [24] (Section 1.2) which utilizes both memory and CPU time to create hash keys for each hash index to every tuple associated with the state. Thus, BI satisfies the low memory and CPU requirement.

Also, adapting BI requires on average less CPU time than the hash index approach due to the number of hash indices supported. To adapt tuples in the state from index BI_1 to BI_2 requires the relocation of each stored tuple to the buckets defined by BI_2, and deletion of the memory associated with BI_1. While the hash index approach may need to create and delete multiple hash keys for each stored tuple.

Figure 4. State using a Bit Address Index



bit-address index
Partition Address
A1
A2
A3
Address Book
0
1
…
1023
Bucket 0
Bucket 1
Bucket 250
Bucket 1023
attributes
A1
A2
A3
insert tuple
2012
125-89
47
hash(A1) = hash(2012) = 7 = 00111
hash(A2) = hash(125-89) = 3 = 11
hash(A3) = hash('47') = 2 = 010
bucket_addr = 0011111010 = 250
attributes
A1
A2
A3
search request
2012
*
47
hash(A1) = hash(2012) = 7 = 00111
hash(A2) = hash(*) = 00 ~ 11
hash(A3) = hash('47') = 2 = 010
bucket_addr1 = 0011100010 = 226
bucket_addr2 = 0011101010 = 234
bucket_addr3 = 0011110010 = 242
bucket_addr4 = 0011111010 = 250

Next, we compare the performance of searching for tuples using BI in Figure 4 versus using the hash index approach in Figure 1. Consider search request sr_1 looking for all packages with priority code $= 2012$ and location id = '47' (Figure 4). First the bucket ids that must be searched are found by mapping the attributes specified in sr_1 (i.e., 00111 and 010) and the attributes not specified in sr_1 (i.e., attributes represented by the wild card symbol $*$ or in the case of sr_1, $t.A2$ whose bit values cover 00 to 11). Then the bits are combined into bit strings (i.e., 0011100010, 0011101010, 0011110010, and 0011111010). Finally a scan is performed across all identified bucket(s) (i.e., buckets 226, 234, 242, and 250). If the search is narrow and precise (i.e., the access pattern of the search request specifies all join attributes) then only one bucket will need to be searched. If the search is wide (i.e., the access pattern contains wild card symbols) then several buckets will need to be searched. On average we expect a good index configuration to be able to limit the number of buckets required to be scanned for the majority of search requests. Clearly, a single BI can serve multiple access patterns and require less memory and CPU for maintenance than multiple hash keys. Thus BI satisfies the diverse access pattern requirement (Section 1.4).

The configuration of the index key map (i.e., the assignment of which attribute data values map to which bits for bucket addresses) influences the number of buckets that must be evaluated during the search. The optimal index key map is configured so that there exists a relatively even distribution of stored tuples among the buckets. This requires the analysis of the distributions of the data content of each attribute used to index stored tuples. Index key map selection is a generic hashing issue not specific to AMR systems.

Inequality Predicates: To handle inequality predicates requires the minor adjustment of locating each tuple whose attribute is not equal to the search request. For such an inequality query (which is less common than equality queries in the streaming context), either state can be scanned if the number of equal values is small. Or, alternatively, the equality bits from the attributes in the search request can be computed. Then all other buckets besides those indicated by the hash must be searched. To simplify the presentation, equality predicates are used in the rest of our discussion.

3.2 Multiple Index Configurations

The query paths used to process new search requests in AMR systems are constantly evolving. As the query paths evolve, the system will still contain some older partially serviced search requests already in process. These older partially serviced search requests may favor a different query path than newly incoming search requests. To minimize query response time, states must efficiently process all search requests.

Traditional on-line index tuning approaches support a single common index structure for all search requests regardless of their proximity to completion. In such systems, when a new index structure is chosen all stored tuples in a state are migrated into the new index structure. In AMR systems, a single index configuration may not cover a significant portion of the join predicates of both newly incoming and partially serviced search requests as they may be routed along different query paths. Hence to support such diversity of search requests, we propose to allow states to maintain multiple index configurations where each index configuration represents a time span of tuples such that distinct index configurations can exists for older and newer search requests.

Every search request sent to a join operator regardless of its age joins with tuples stored in the state that are within the query window of the search request. Stored tuples could be time-partitioned according to the search requests that they support. Then each partition of stored tuples could be indexed to support efficient processing of the search requests that they provide results for (e.g., indexed to efficiently support the query paths of older search requests).

Partition Design: To meet these requirements, we associate a variable with each index configuration that represents the query window into which all tuple(s) that are stored under it fall. Hence the tuples stored in the state are broken into time slices which we refer to as partitions *P*. A partition is a portion of the query window of constant length $|P|$.

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 from a given stream received thus far. A query window QW is composed of a number of partitions denoted by PW, where $PW = \lceil\frac{QW}{|P|}\rceil$. Each index configuration is stored in a hash table where the partition number is the key. To reduce the amount of memory required, consecutive partitions that use the same index configuration are stored in the same BI.

4 Index Assessment

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

4.1 Index Configuration Dependent Costs C_D

The quality of an index configuration IC depends upon the estimated total *unit processing cost* for IC [18] which corresponds to the combined maintenance and search costs [13]. The maintenance costs are the sum of the costs to insert and delete tuples in a state and to compute bucket ids $(C_{insert} + C_{delete} + C_{hash,I})$. The search costs are the sum of the costs to compute bucket ids and search for tuples stored in the state $(C_{hash,S_r} + C_{search})$. C_{insert} and C_{delete} are independent of which IC is evaluated. Henceforth when comparing different index configurations we only need to consider the index configuration dependent costs C_D (Equation 1) [13]. See notations in Table 1 [13].

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
N_A	# of indexed attributes
$N_{A,ap}$	# 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
A	the set of all search access patterns that arrived within a time unit

$$\begin{aligned} C_D &= C_{hash,I} + C_{hash,S_r} + C_{search} \\ C_D &= (\lambda_d N_A C_h + \lambda_r \sum_{ap \in A} (N_{A,ap} C_h + \frac{\lambda_d W_{ap} F_{ap}}{2^{B_{ap}}} C_c)) \end{aligned} \qquad (1)$$

4.2 Access Pattern Statistics

Assessing possible index configurations requires the collection of statistics on the frequency of each access pattern (f_{ap}).

The frequency of access pattern *ap in a workload D, denoted as f_{ap}, is* $f_{ap} = \frac{A_{ap}}{|A|}$ *where A_{ap} is the number of search access patterns in D for ap and $|A|$ is the total number of search access patterns utilized in D.*

To collect statistics on the frequency of each access pattern the system tracks the total count of search requests received for each possible access pattern. The number of possible access patterns is equal to the number of combinations of join attributes for a given state. If there are N_{ja} join attributes then the number of possible access patterns is $\sum_{k=1}^{N_{ja}} \binom{N_{ja}}{k}$. Thus the number of possible access patterns is exponential in the number of join attributes.

4.3 Self Reliant Index Assessment SRIA

4.3.1 Basic SRIA

We first sketch the basic index assessment algorithm, called *Self Reliant Index Assessment*. SRIA captures the total count of each access pattern received for a given 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 $B(ap)$. A 1 indicates that a particular join attribute is used to search, while a 0 indicates that a join attribute is *not* used to search. Consider a state with 3 join attributes $\{A, B, C\}$. If ap_1 is searching using only attribute A (i.e., ap_1 = $< A, *, * >$) then $B(ap_1)$ = 100 which represents index number 4. If ap_1 is searching using attributes B and C (i.e., ap_1 = $< *, B, C >$) then $B(ap_1)$ = 011 which represents index number 3. Finally if ap_1 is searching using all attributes (A, B, and C) (i.e., ap_1 = $< A, B, C >$) then $B(ap_1)$ = 111 which represents index number 7.

Each incoming access pattern ap is processed using the $B(ap)$ function. If the access pattern exists in SRIA then A_{ap} is incremented by 1. Otherwise the new access pattern ap is created in SRIA with A_{ap} set to 1.

4.3.2 Compact SRIA: Access Pattern Reduction

The large number of possible access patterns (Section 4.2) can cause potential memory limitations. We now explore a frequency access pattern reduction method extension to *SRIA* referred to as *Compact SRIA* or *CSRIA*. *CSRIA* is modeled after the heavy hitter algorithm proposed by Manku and Motwani [21]. Informally, during assessment *CSRIA* periodically removes any access pattern statistic whose frequency falls below a preset error rate or ϵ. Upon completion of assessment, *CSRIA* returns all access pattern statistics whose frequencies are above a preset threshold θ.

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: $\forall ap \in Q : (f_{ap} + \delta) \geq (\theta - \epsilon)$ and $\forall ap \in (A_{ap} - Q) : (f_{ap} + \delta) < (\theta - \epsilon)$*

CSRIA works by evaluating incoming access patterns in segments. Each segment contains $\lceil \frac{1}{\epsilon} \rceil$ search requests. Segments are associated with an 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 or s_{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 access patterns are not deleted too early, the current maximum error in frequency δ is stored with each f_{ap} statistic. The maximum error in frequency δ represents the minimum number of requests that should have occurred in order for f_{ap} to be above the preset error rate threshold.

CSRIA is composed of three phases, *insertion* (creating statistics), *compression* (removing statistics), and *final results* (finding all statistics that meet the threshold). *Insertion* and *compression* occur during the assessment phase, namely, assessment collects compact statistics for a predefined period of time. During assessment, creating 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 already exists in SRIA then the frequency A_{ap} is increased by 1. Otherwise a new access pattern is created in SRIA with A_{ap} equal to 1 and the maximum error in frequency collected δ equal to $s_{id} - 1$. Also during assessment, *compression* is executed whenever a segment worth of search requests has been received. *Compression* removes any access pattern from the SRIA table whose frequency is below the preset error threshold, i.e., $A_{ap} + \delta \leq s_{id}$. At the end of assessment, the *final result* is produced by locating any access pattern whose $f_{ap} + \delta$ is greater than the preset threshold in accordance with the preset error rate, i.e., $f_{ap} + \delta \geq \theta - \epsilon$.

Some benefits to this approach are: 1) It guarantees to find any access pattern whose frequency is greater than the preset threshold θ. 2) The memory required is limited to at most $\frac{1}{\epsilon}log(\epsilon \sum_{m=0}^{N_{ja}-1} \binom{N_{ja}-1}{m})$ access patterns where N_{ja} is the number of join attributes [16].

Table 2. Compact SRIA Estimation Example

ap	f_{ap}	ap	f_{ap}
<A, *, *>	4%	<A, B, *>	4%
<*, B, *>	10%	<A, *, C>	16%
<*, *, C>	10%	<*, B, C>	10%
		<A, B, C>	46%

Discussion: CSRIA, while efficient, fails to utilize the relationship 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 index configuration. If θ is 5% and ϵ is .1%, then *CSRIA* will delete access patterns $< A, *, * >$ and $< A, B, * >$ even though both access patterns would benefit from an index configuration containing $< A, *, * >$. Furthermore the combined frequency of $< A, *, * >$ and $< A, B, * >$ is 8% which is greater than θ. The index configuration found during selection is the configuration with 1 bit assigned to the B attribute and 3 bits assigned to the C attribute. Whereas the optimal index configuration is the configuration with 1 bit assigned to A and B attributes each and 2 bits assigned to the C attribute.

4.4 Dependent Index Assessment DIA

4.4.1 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 the dependent access pattern relationships can be used in a sophisticated index assessment approach referred to as *Dependent Index Assessment* or DIA.

A search request with access pattern ap_i will be more efficiently executed if an index exists such that the combination of join attributes supported by the index is a subset of the join attributes in ap_i as compared to an index that includes join attributes not in ap_i. Consider a state with $JAS = \{A, B, C, D\}$ and a search request with $ap =< A, B, *, * >$:

In the *worst case*, IC consists of no attributes in ap (e.g., IC consists of attributes C and D). As IC and ap have no common attributes, no buckets can be eliminated via matching a search request attribute to a specific bucket. Thus a full scan is required (i.e, a comparison of all tuples stored in the state).

In a *slightly better case*, IC consists of some attributes in ap as well as some attributes not in ap (e.g., IC consists of attributes A, B, and C). Each attribute in the IC that is not in ap creates the search wild card condition described in Section 3.1. If n is the number of bits assigned to the attributes not in ap then 2^n buckets will need to be compared.

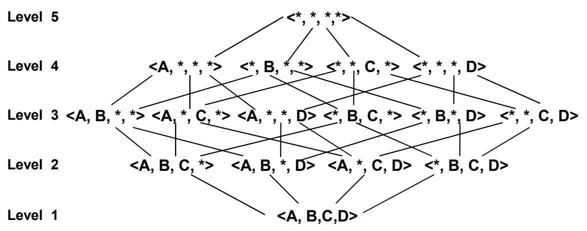
In a *much better case*, the attributes in the IC are a subset of the attributes in ap (e.g., IC consists of attribute A). In this case, one single bucket worth of tuples must be searched. The bucket searched will contain all tuples relevant to ap as no wild card condition exists but not every tuple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

In the *optimal case*, the attributes in the IC exactly match the attributes in ap (e.g., IC consists of attributes A and B). 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, the number of tuples required to be compared is the smallest number of tuples that ap would be required to search for using this IC.

Definition 1 *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 relationship organizes the access patterns into a lattice (Figure 5). Each node in the lattice corresponds to an ap. The lattice is formed by starting with a single node representing the ap that contains no join attributes (top node). At each level in the lattice, nodes are formed by taking each node in the prior level and adding one join attribute not already in it. This process continues until the final level where all possible join attributes are included in a single node (bottom node). Nodes from one level are linked to nodes that they provide a *search benefit* to in the level directly below as represented by lines in Figure 5.

Figure 5. State with 4 join attributes



Dependent Index Assessment DIA stores the assessment values in a lattice to retain the dependent search benefit relationships between access patterns. Rather than starting with a complete lattice, DIA builds a lattice in a top-down manner at runtime. Each node N in the partial lattice L consists of the access pattern it represents, namely $N.ap$, and the count of $N.ap$ requests, or $N.A_{ap}$.

For each search request, if a node exists in the lattice that matches the access pattern then the corresponding count $N.A_{ap}$ is incremented by 1. Otherwise, a new node is created for the access pattern. To enable quick direct referencing to the access patterns, each node in the *DIA* lattice is mapped to a unique binary representation in the same fashion as outlined above for $SRIA$. As such, physically each *DIA* node is stored in a *SRIA* table.

4.4.2 Compact DIA: Access Pattern Compression

The large number of possible access patterns in DIA (Section 4.2) can cause memory limitations. We now explore an access pattern reduction method extension to *DIA* modeled after a hierarchical heavy hitter algorithm proposed by Cormode et al. [10] namely *Compact DIA*, or *CDIA*. In our context, the search benefit relationship is utilized to combine access pattern statistics rather than deleting them. During assessment, *CDIA* periodically combines the statistics of any access pattern ap whose frequency falls below a preset error rate ϵ with the statistics of any access pattern that provides search benefits to ap. At the end of assessment, *CDIA* returns all access pattern statistics whose frequencies are above a preset threshold θ.

Given a state S_t, DIA access patterns, threshold θ, error rate ϵ, the set of all possible access patterns of S_t referred to as P_{AP}, the CDIA *algorithm outputs the set of frequency access patterns Q such that: $\forall ap \in Q$: ${f^*}_{ap} - \epsilon \leq f_{ap} \leq {f^*}_{ap}$ where $({f^*}_{ap} = \sum f_k : (k \in P_{AP})\, and\, (ap \prec k))$ and $\forall\ ap \notin Q$: $\sum f_k \leq \theta$ where $(\forall k \in P_{AP}\, (k \notin Q)\, and\, (ap \prec k))$.*

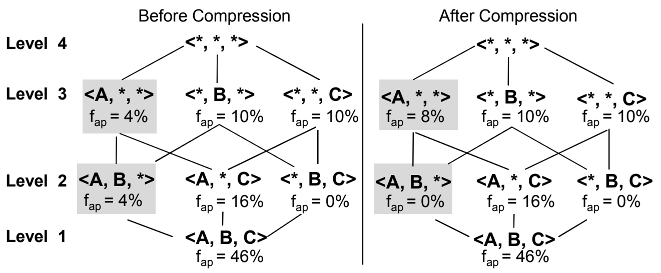
The *CDIA* approach provides three methods *insertion* (creating statistics), *compression* (combining statistics), and *final results* (finding all statistics that meet the threshold). The *insert* method is composed of multiple steps. First, the binary representation of the access pattern $B(ap)$ is computed. Then, if a node exists in the lattice representing ap the count of this access pattern A_{ap} is increased by 1. Otherwise, a new node is created for access pattern ap with A_{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 lattice. A leaf node is any node in the lattice that does not provide a search benefit to any other node (i.e., no node below it has a count > 0). *Compression* proceeds as follows. For each leaf node in the lattice, if the total count of the given access pattern A_{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 A_{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* computation step, any access pattern whose frequency is greater than θ is identified and returned as described below. For each node N in the lattice starting from the nodes in the lowest level of the lattice, first the frequency of the access pattern $N.f_{ap}$ is computed. If $N.f_{ap}$ is less than θ, then a parent of the leaf node is identified, and the access pattern count of the leaf node is added to the parent. Otherwise, the node is added to the result set Q.

CDIA Combination Methods: Several strategies can be utilized to combine the frequency of a child node N_c with that of a parent node N_p that provides search benefits to it [11]. 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).

There are several benefits to the CDIA approach: 1) It guarantees to find any access pattern 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 equal to the number of levels in the lattice [10]. 3) It reduces the number of access patterns stored while retaining the statistics of removed access patterns.

Figure 6. CDIA Example



Reconsider the example in Section 4.3.2 with a state containing 3 join attributes, a 4 bit index configuration, and DIA outlined in Figure 6. If θ is 5% and ϵ is $.1\%$, the CDIA approach using the random combination method combines the frequency of $< A, B, * >$ into $< A, *, * >$. In this case, index selection will find the true optimal index configuration.

5 Bounds on Statistics Reductions

Index selection finds the best IC for each state by comparing the IC dependent costs C_D (Equation 1) of each possible configuration in the state and returning the IC with the lowest combined cost C_D. To achieve this the "optimal" division of available bits among the states that minimizes the combined cost C_D must be considered. The bit division problem given the constraint of minimizing combined dependent costs C_D across all states has been shown to be equivalent to the multiple-choice knapsack problem (MCKP) [23]. MCKP has been proven to be NP-hard. Therefore we consider instead the problem of locating the optimal IC for each state where each state is allocated a set number of bits. Henceforth *index selection* refers to the problem of finding the best IC for a state among all possible ICs given the allocation of B bits (Equation 1), i.e., returning the IC with the lowest cost C_D.

5.1 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 $\leq Min(N, B)$ attributes is explored. For each IC with bits assigned to k of the N join attributes there are $\begin{pmatrix} B-1 \\ k-1 \end{pmatrix}$ possible combinations of distributing B bits among the k attributes. Hence the complexity of the search space is $\sum_{k=1}^{Min(N,B)} \begin{pmatrix} N \\ k \end{pmatrix} \begin{pmatrix} B-1 \\ k-1 \end{pmatrix}$. As the size of both N and B increases, the search space grows exponentially.

Prior to index selection we use the *EPrune* algorithm [13] to remove attributes for which the indexing overhead exceeds the cost reduction gained. EPrune guarantees to find the optimal IC [13]. It quickly finds the optimal IC when the number of join attribute statistics N is small. Complementary to this, our *CSRIA* and *CDIA* methods assist in minimizing the search space by reducing the number of frequency access pattern statistics. This increases the number of possible attributes eliminated by EPrune.

As indicated in Section 4.2, the number of cost calculations performed is driven by the number of access pattern statistics. Given a large number of access pattern statistics, the number of cost calculations performed to evaluate C_D for each possible IC can also be large and thus costly. *Eliminating a single access pattern statistic per the* CSRIA *and* CDIA *methods reduces the number of cost calculations performed by* $\sum_{k=1}^{Min(N,B)} \begin{pmatrix} N \\ k \end{pmatrix} \begin{pmatrix} B-1 \\ k-1 \end{pmatrix}$, *resulting in significant savings.*

5.2 Statistic Reductions

CSRIA and *CDIA* reduce the number of frequency access patterns statistics collected with the goal of reducing the amount of memory required for index assessment as well as the amount of CPU cycles required for index selection. However, such reductions clearly could influence the quality of the identified IC. *We now show the important result that the affect of removing a frequency access pattern on* C_D *is bounded in proportion to the preset threshold* θ *and error rate* ϵ. An access pattern statistic F_{ap} can have one of the following affects on a given IC's dependent cost C_D (Equation 1):

If each attribute in the access pattern ap is in the IC and visa versa, then all possible bits are used to search (i.e., no wild cards). The portion of C_D that the frequency of ap F_{ap} contributes to is $\lambda_r \frac{\lambda_d W_{ap} F_{ap}}{2^B} C_c$ or 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 tuples stored in the state is required. The portion of C_D that F_{ap} contributes to is $\lambda_r \lambda_d W_{ap} F_{ap} C_c$ or 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, namely $\lambda_r \lambda_d W_{ap} F_{ap} C_c$, from each possible C_D calculation. Thus any access pattern whose frequency is $\leq (\theta - \epsilon)$ is guaranteed to not reduce any index configuration dependent costs 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 $\leq (\theta - \epsilon)$ is guaranteed not to increase any index configuration dependent costs C_D by more than $\lambda_r \lambda_d W_{ap} \theta C_c$.

6 Scheduling The Tuning Process

We now introduce four scheduling methods for determining when to initiate index tuning.

Continuous Index Tuning: Continuous Index Tuning collects statistics continuously on each state for a predetermined period of time. Upon completion of the preset evaluation window, index selection is executed and assessment is restarted. In this case when the access patterns of the search requests are not changing, this method adds significant overhead for assessment even though no new index configuration is required.

Periodic Index Tuning: Adapted from index tuning in traditional database systems [15], during periodic time intervals statistics are collected and used to search for the best index configuration. The downfall of this method is when the access patterns of search requests are rapidly changing, this method will not identify changes in a given state that may arise while the state is not being assessed.

Triggered Index Tuning: Triggered Index Tuning is based upon observed changes in routing. It initiates index assessment on states with observed significant shifts in routing. In this method, over the period of a single state index assessment the number of search requests sent to each state is tracked and compared. The state with the most significant change is triggered for index assessment. A significant change in the number of search requests is measured by computing the degree of difference between the current number of search requests for a given state S_t to the prior number of search requests for S_t. This method tends to more rapidly identify significant changes in the number of search requests for a given state than the periodic index tuning method. The only drawback of this strategy may be that states that never have significant changes in the number of search requests may never be assessed.

Hybrid Index Tuning: This leads us to propose the hybrid of employing both periodic and triggered methods. An ordered list of the states to be assessed is kept. When a state has been assessed due to triggered index tuning, the state is moved to the bottom of the assessment list. The hybrid is the method of choice since it allows the system to catch significant changes in individual states as they occur while ensuring that each state is still periodically assessed.

7 Index Migration

Once a new ”best” index configuration has been discovered for a state, index migration considers whether or not to re-index a state from the current index configuration(s) to this new index configuration. We now present three index migration methods, namely *All*, *Nothing*, and *Partial*.

7.1 All or Nothing

Hence the *all index migration* re-indexes *all* the tuples currently stored in the state to the new ”best” index configuration, and removes the old index configuration(s). The result of this approach is that states support a single common index configuration at any given time.

In contrast the *nothing index migration* does not re-index any of tuples currently stored in the state (i.e., all the tuples currently stored in the state are kept in their current index configurations). As a result, states must support multiple distinct index configurations at the same time. However any newly arriving tuple will only be indexed using the new ”best” index configuration.

Analysis: Is a single index configuration better than multiple distinct index configurations? To answer this question we must explore the overhead of index migration. Both approaches carry no migration decision overhead as no evaluation is performed. To support multiple distinct index configurations requires no additional migration cost as no migration is required. While to support a single index configuration requires all tuples stored in the state to be migrated to the new index configuration. The cost to migrate a single tuple is the sum of the costs to compute 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 4). The cost to remove a single stored tuple from an index configuration is the sum of the costs to compute the memory location of the tuple using the index configuration, and to delete the tuple $(C_{hash,R} + C_{delete})$. Thus the total cost of migration using a single index configuration is $\sum\limits_{t \in state} (C_{insert} + C_{hash,I}) + (C_{hash,R} + C_{delete})$.

In order for the cost of migration 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 (Equation 2). If $B_m(IC, BIC)$ is greater than 0 then there is a benefit in migrating IC to BIC.

$$\begin{aligned} B_m(IC, BIC) &= C_s(IC) - \\ &\quad (C_s(BIC) + C_m(IC, BIC)) \\ C_s(IC) &= C_{hash,R} + C_{probe} \\ C_m(IC, BIC) &= \sum_{t \in state} ((C_{hash,I} + C_{insert}) + \\ &\quad (C_{hash,R} + C_{delete})) \end{aligned} \tag{2}$$

Thus in order for *All index migration* to have a lower overall processing cost than *Nothing index migration*, the cost to search generated from older partially serviced search requests using the new "best" index configuration plus the cost of migration must be less than the cost to search generated from older partially serviced search requests 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 older search requests using their current index configuration and utilize the new "best" index for only newly incoming search requests.

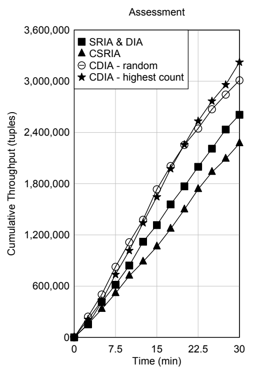
7.2 Partial Migration

If multiple index configurations are supported as explained above, then the question arises, is there a benefit to migrate only some of them selectively. *Partial index migration* supports multiple index configurations, and selectively migrates individual index configurations based upon the estimated future workload of partially serviced search requests. We estimate the future workload of each partition by tracking for each partition the number of active partially serviced search requests within its query window (Section 3.2). For each time-sliced index configuration in the state, the benefit of migration $B_m(IC, CIC)$ using the estimated number of future older partially serviced search requests served by the given partition is evaluated (i.e., when evaluating $C_s(IC)$ and $C_s(BIC)$ the estimated number of search requests received within a time unit λ_r is replaced by the number of active partially serviced search requests within the partition's query window). Only tuples stored in partitions where there is a benefit in migrating are migrated.

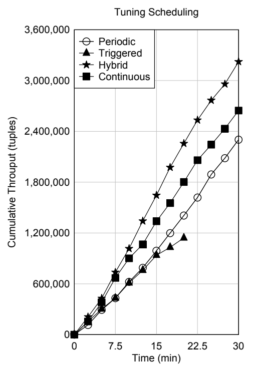
Partial index migration adds the following additional costs to processing: 1) the cost to evaluate each partition every time a new index configuration is selected, and 2) the cost for the router to track the estimated future workload of older partially serviced search requests. Thus in order for *Partial index migration* to have a lower overall processing cost than *Nothing index migration*, cost to search generated from older partially serviced search requests using the new "best" index configuration plus the cost of migration and the cost to evaluate the migration needs of each partition must be less than the cost to search generated from older partially serviced search requests using the current index configuration(s).

8 Experimental Results

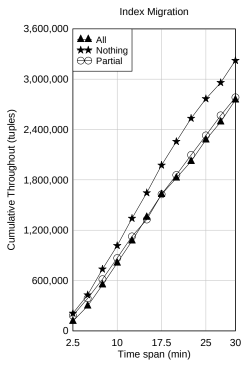
Experimental Setup: All experiments are implemented in the CAPE stream system [28] using Linux machines with AMD 2.6GHz Dual Core CPUs and 4GB memory. They compare throughput defined as the cumulative output tuples produced over a fixed period of time. Our experimental study explores: 1) Which of our proposed alternative tuning methods is the most effective at improving the throughput in AMR systems?, 2) Is AMRI more effective than state-of-art approaches at improving throughput?, and 3) How does varying the number of possible query paths affect the throughput of an AMR system using our proposed index solution versus the state-of-art approach?



a) Index Assessment



b) Index Selection



c) Index Migration

Figure 7. Evaluating Alternative Tuning Methods

Every experiment uses a 4 way join query across 4 data streams. Every stream is joined to each of the 3 other streams via a unique join attribute. Thus each state contains 3 join attributes. Each state is required to efficiently support search requests containing all possible combinations of the 3 join attributes (7 possible access patterns). Our results illustrate that even for systems with a small number of possible access patterns, there already is a significant benefit in removing access pattern statistics. Clearly, as the number of possible access patterns in a state increases so does the probability of access pattern statistics being eliminated.

Synthetic Data Sets: To test the effectiveness of our methods under conditions where adaptive indices are required we created synthetic data in which the selectivities of joining a stream to any of the other streams vary periodically. This may cause the router to use new query paths which in turn may initiate index selection.

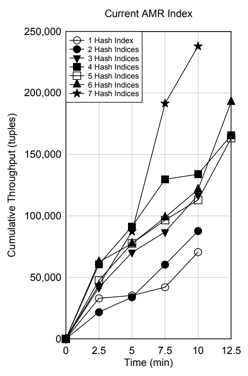
Real Data Sets: We use real data collected from 54 sensors deployed in the Intel Berkeley Research lab between February 28th and April 5th, 2004 (http://berkeley.intel-research.net/labdata/). The sensors were divided into 4 streams based upon their location. The query locates sensor data from the 4 streams where the temperature, humidity and relative location (horizontal distance from a fixed point) is the same. This query would be useful to an engineer monitoring a building's environment.

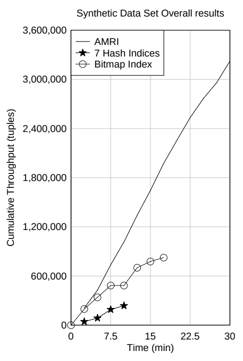
The IC on each state uses 64 bits and is initiated by running index selection using statistics gathered by executing the stream for 15 minutes (as quasi training data). For the state-of-the-art approach, the starting indices are those found to support the most frequent search request access patterns by running the quasi training data approach above. Unless specified assume all experiments use the synthetic data set and query.

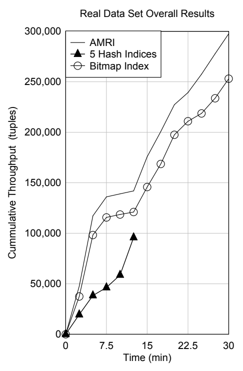
8.1 Evaluating Alternative Tuning Methods

Index Assessment: We now 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 (Figure 7 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) prior to index selection. Note that the results of DIA and SRIA are equal. This is not surprising as they share the same code base, use the same SRIA table, and do not reduce any nodes.

Index Tuning Scheduling: Next, we compare index tuning scheduling methods *Periodic*, *Triggered*, *Hybrid*, and *Continuous* (Section 5). Each test uses the CDIA with highest count compression set up outlined above (i.e., the most effective method shown above). *Hybrid* outperforms *Periodic*, *Triggered*, and *Continuous* Index Tuning







a) State-of-art AMR Index b) State-of-art vs AMRI using Synthetic Data c) State-of-art vs AMRI using Real Data

Figure 8. AMRI vs State-of-Art Experiments

by 65%, 29%, and 18% respectively (Figure 7 b). This shows the utility of combining traditional periodic index tuning with knowledge gained by the router (i.e., *Hybrid* approach). *Triggered* ran out of memory after 20 minutes. In triggered index tuning scheduling, states with no significant changes are never assessed. Thereby such states have suboptimal index configurations which increases the time to process search requests and in turn delays the processing of incoming search requests.

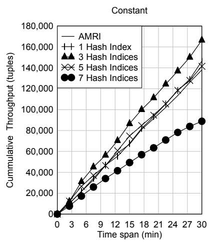
Index Migration: Next, we study the impact of runtime index migration methods (Section 7). Each method is run using the *Hybrid* index selection set up outlined above (i.e., the most effective method shown above). *Nothing* index migration (i.e., utilizing multiple time-partitioned non migrating index configurations) outperforms both *All* and *Partial* index migration by 19% (Figure 7 c). The overhead of migrating the current index configuration to a single new common index configuration (i.e., *All* Index Migration) is greater than the actual search execution time saved. In contrast to traditional systems that support a single common index design for the entire set of stored tuples, 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* index migration (i.e., utilizing a single common index configuration) would have outperformed *Nothing* index migration. We also conclude that the overhead to migrate selective index configurations (i.e., *Partial* index migration) is greater than the actual search execution time saved.

8.2 AMRI vs State-of-Art

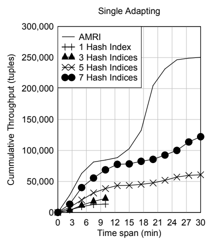
Synthetic Data: Next, we evaluate the state-of-art AMR indexing (i.e., multiple hash indices) [24]. This experiment varies the number hash indices on each state from the minimum to maximum number of possible indices, 1 to 7 respectively. Static non-adapting hash indices (i.e., no index tuning) produced poor results. Thus adaptive hash indices using the nothing index migration set up outlined above and conventional index selection (i.e., indices created support the most frequent search request access patterns) are used.

No trial ran for more than 12.5 minutes (Figure 8 a). In each case, the system ran out of memory due to the large amount of CPU time and memory overhead required to maintain the indices (See Section 3.2). For systems with only a few indices a backlog of active search requests occurs from the processing delay caused by the large number of complete scans performed. AMRI emerges as the clear winner that is AMRI produces 93% more results than even the best hash index configuration (Figure 8 b).

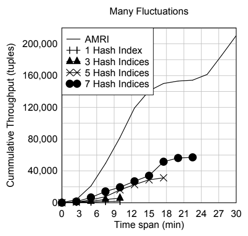
We now compare our AMRI index tuning to the ”state-of-the art” static bitmap index. Both start with the same optimal index configuration. The non-adapting bitmap index could not keep up with the search requests and ran out of memory after 15.5 minutes. AMRI produces 75% more results than the non-adapting bitmap index (Figure 8 b).



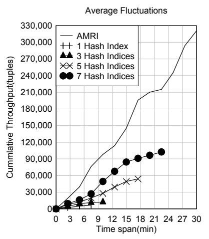
a) Constant



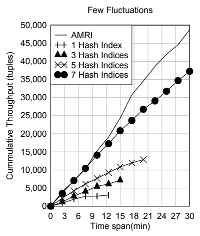
b) Single Adapting



c) Many Fluctuations



d) Average Fluctuations



e) Few Fluctuations

Figure 9. Skew on the Number of Query Paths Experiments

Real Data: Next, we compare AMRI to the state-of-art AMR indexing as in [24] using the Intel Berkeley Research lab data set and query outlined above. AMRI implements CDIA using highest count compression index assessment where the maximum error $\delta = .1$, threshold $\theta = .15$, hybrid index tuning scheduling, and nothing index migration. The same amount of memory is available to both the AMRI and hash indices systems.

We again observed very poor results using non adapting hash indices. Thus adaptive hash indices are implemented using the manner outlined above. AMRI again produces more results than the best hash index solution (68% more) (Figure 8 c). Next we compare AMRI to a non-adapting bitmap index. Both start with the same index configuration. In this case study, AMRI produces 15% more results than the non-adapting 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.

8.3 Skewing the Number of Possible Query Paths

We now compare AMRI to adaptive hash indices as defined above in systems with varied 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 index tuning scheduling, and nothing index migration. 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.

In the *Constant* experiment (Figure 9 a), only 1 non adapting query path for each stream is selected by the router (i.e., 1 query path is optimal for the duration of the query). 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. But such a stable a 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.

In the *Single Adapting* experiment (Figure 9 b) we considered the case where the router selects a single path for each stream and 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. This supports our claim that AMRI improves the throughput of queries in fluctuating environments.

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 (Figure 9 c, d, & 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 (Figure 9 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 (Figure 9 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 (Figure 9 c). Clearly, AMRI adapts to systems with higher fluctuations better than the state-of-art approach.

9 Related Work

The first AMR system, Eddy, used a router to route individual tuples through operators [6]. Enhancements in routing policies were proposed in [7, 31]. Multi-query AMR system issues were covered by [20]. [24] improved optimization capabilities by extending routing flexibility through the development of the STeM operator, a unary join operator, and access modules, an index over data from a single stream stored in a state. As shown in our experiments the overhead of access modules makes such hash indices ineffective for AMR systems.

Index Tuning in static databases aims to find a set of indexes that maximally benefit a given query workload by either selecting the most optimal index configuration off-line [3, 9, 2, 12] or online during execution [4, 29, 8]. Online index tuning continuously evaluates and adjusts the index configuration to the current workload. Our solution borrows from the online index tuning work. Online tuning in static databases requires the system indices to adapt to observed changes in the workload [4, 29, 8]. We explore novel index tuning approaches that reduce the system resources required while maintaining the integrity of the index selected in accordance with a preset threshold and error rate.

Bit-address indexing, initially designed to index partial match queries on a file database [5], has been applied on applications ranging from compactly storing very large multidimensional arrays [25] to reducing processing costs of multidimensional queries [26, 27]. [13] studied index selection heuristics using a single bit-address index where the search request workload is known prior to execution. We tackle the on-line index tuning problem for AMR systems.

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. We put forth in our work that heavy hitter algorithms, a type of stream sampling algorithm, meet the requirements of AMR systems (Section 5) as they analyze and report all items that appear above a preset threshold [22, 19, 21].

[22] introduced the first deterministic algorithm for approximating frequency counts, called the heavy hitter method. [21] added the error rate ϵ approximation guarantee. Our *CSRIA* method is modeled after the heavy hitter algorithm proposed in [21]. Hierarchical heavy hitter, applying the heavy hitter methodology to hierarchical multi-dimensional data, was studied in [10, 14] 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 [11]. We customize the compression methods to handle the search benefit relationships between indices in AMR systems. To our knowledge, ours is the first application of heavy hitter and hierarchical heavy hitter algorithms to address the problem of index tuning in AMR systems.

10 Conclusions

We developed the Adaptive Multi-Route Index for AMR systems or AMRI, in short. AMRI employs multiple time-partitioned bitmap indices to serve a workload composed of diverse query access patterns with partially overlapping life spans. We propose AMRI tuning methods, in particular, four index assessment methods (SRIA, CSRIA, DIA, and CDIA), four index tuning scheduling methods (Periodic, Triggered, Hybrid, and Continuous), and three index migration methods (All, Nothing, and Partial). Bounds on the optimality (i.e., quality) of the index configuration found during index selection are also established in this work.

Our experimental study demonstrates the overall effectiveness of AMRI at improving throughput in dynamic stream environments while keeping the index tuning costs to a minimum. In particular, using synthetic data AMRI produced on average 93% more results than the state-of-art approach and in our case study with real data on average 68% more results over the same period of time. We clearly demonstrated that compared to All index migration (i.e., a state utilizing a single index configuration), Nothing index migration (i.e., a state utilizing a multiple time spliced index configurations) significantly improved the throughput of AMR systems.

11 Acknowledgments

We thank Luping Ding, and WPI graduate peers for revisions and CAPE software. This work is supported by the following grants: NSF 0917017, NSF CNS CRI 0551584 (equipment grant), NSF 0414567, and GAANN.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB*, pages 120–139, 2003.

[2] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005: demo. In *SIGMOD*, pages 930–932, New York, NY, USA, 2005. ACM.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[4] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD*, pages 683–694, New York, NY, USA, 2006. ACM.

[5] A. V. Aho and J. D. Ullman. Optimal partial-match retrieval when fields are independently specified. *ACM Trans. Database Syst.*, 4(2):168–179, 1979.

[6] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD*, 29(2):261–272, 2000.

[7] P. Bizarro, S. Babu, D. DeWitt, and J. Widom. Content-based routing: different plans for different data. In *VLDB*, pages 757–768, 2005.

[8] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. *ICDE*, 0:826–835, 2007.

[9] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[10] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding hierarchical heavy hitters in data streams. In *VLDB*, pages 464–475. VLDB Endowment, 2003.

[11] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Diamond in the rough: finding hierarchical heavy hitters in multi-dimensional data. In *SIGMOD*, pages 155–166, New York, NY, USA, 2004. ACM.

[12] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *VLDB*, pages 1098–1109. VLDB Endowment, 2004.

[13] L. Ding and E. A. Rundensteiner. Index tuning for parameterized streaming groupby queries. In *SSPS*, pages 68–78, New York, NY, USA, 2008. ACM.

[14] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.

[15] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *SIGMOD*, pages 1–8, New York, NY, USA, 1976. ACM.

[16] T. Johnson, S. Muthukrishnan, and I. Rozenbaum. Sampling algorithms in a stream operator. In *SIGMOD*, pages 1–12, New York, NY, USA, 2005. ACM.

[17] W. K., R. E., and A. E. Index tuning for adaptive multi-route data stream systems. In *IEEE SSPS*, 2010.

[18] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. *ICDE*, 00:341, 2003.

[19] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.

[20] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60, New York, NY, USA, 2002. ACM.

[21] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357. VLDB Endowment, 2002.

[22] J. Misra and D. Gries. Finding repeated elements. Technical report, Ithaca, NY, USA, 1982.

[23] A. Mohr. Bit allocation in sub-linear time and the multiple-choice knapsack problem. In *Data Compression Conference*, page 352, 2002.

[24] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. *ICDE*, 00:353, 2003.

[25] D. Rotem, E. J. Otoo, and S. Seshadri. Chunking of large multidimensional arrays. Technical Report LBNL–63230, Research Org Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), February 2007.

[26] D. Rotem, K. Stockinger, and K. Wu. Towards optimal multi-dimensional query processing with bitmapindices. Technical Report LBNL–58755, Research Org Ernest Orlando Lawrence Berkeley National-Laboratory, Berkeley, CA (US), September 2005.

[27] D. Rotem, K. Stockinger, and K. Wu. Minimizing i/o costs of multi-dimensional queries with bitmap indices. In *SSDBM*, pages 33–44, Washington, DC, USA, 2006. IEEE.

[28] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. CAPE: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

[29] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. Colt: continuous on-line tuning. In *SIGMOD*, pages 793–795, New York, NY, USA, 2006. ACM.

[30] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. pages 459–468, April 2007.

[31] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344. VLDB Endowment, 2003.

[32] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. *SIGMOD*, 27(2):130–141, 1998.