Index Tuning for Adaptive Multi-Route Data Stream Systems

Karen Works
Worcester Polytechnic Institute
line 2: Worcester, MA USA
line 3: Email: kworks@cs.wpi.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute
line 2:Worcester, MA USA
line 3:Email: rundenst@cs.wpi.edu

Emmanuel Agu
Worcester Polytechnic Institute
line 2:Worcester, MA USA
line 3:Email: emmanuel@cs.wpi.edu

***Abstract*—Adaptive multi-route query processing (AMR) is an 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 Adaptive Multi-Route Index (AMRI) employs a bitmap design. Our AMRI design is both versatile in serving a diverse ever changing workload of multiple query access patterns as well as lightweight in terms of maintenance and storage requirements. In addition, our AMRI index tuner exploits the hierarchical interrelationships between query access patterns to compress the statistics collected for assessment. Our experimental study using synthetic data streams has demonstrated that AMRI strikes a balance between supporting effective query processing in dynamic stream environments while keeping the overhead to a minimum.**

I. Introduction

The number of monitoring applications has soared [1]. 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). Consider the stock market. A few years ago, an analyst looking to trade stock may have only considered the current price and volume of that stock. Today, the same analyst needs to combine current price and volume data with the latest company and sector information. Company and sector information is now present in multiple sources (e.g., 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 applications must efficiently function over long periods of time in environments susceptible to frequent fluctuations in data arrival rates [2]. 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 [3]. 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) [3], [4]. AMR systems adapt the order in which operators are executed (i.e., the query path) to current system statistics. The prominent AMR system, Eddy [3], utilizes a central routing operator that decides for (sets of) tuples which operator to visit next based upon the system environment.

While AMR systems adapt query paths, the operators never change. Join operators in AMRs process search requests that can be routed along different query paths before arriving at the operator. The query path taken by a search request determines which tuples constitute the search request. The tuples that embody a search request define the search criteria used to locate tuples in a given state. Consider two tuples t_1 and t_2 from $StreamA$. Tuple t_1 is first routed to join with tuples from $StreamB$ and then to join with tuples from $StreamC$. In contrast 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 between $StreamA$ and $StreamC$ and between $StreamB$ and $StreamC$ (i.e., tuple t_1). AMR systems cannot afford to create indices for every possible query path given the complexity of the number of queries and joins. Therefore when fluctuations occur and the query paths change, it is important for AMRs to tune the index configuration to optimally serve the query path(s) in use. No index tuning solution has been proposed for AMR systems. This is now the focus of our work.

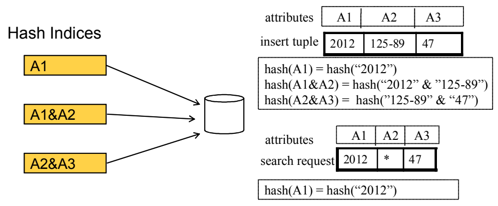
A. Indexing in AMR Systems

Raman et. al. [5] proposed multiple access modules for each state where a state stores tuples originating from a stream. Each access module employs a hash index to optimize a particular data access. We now illustrate possible inefficiencies of access module.

Example: Consider a package tracking 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 DSMS. 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 in 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.

Figure 1. Indexing in AMR Systems



Consider search request sr_1 looking for all packages with priority code $A1$ = 2012 and location id $A3$ = 47. Executing sr_1 involves determining the most suitable hash index for processing sr_1. The most suitable hash index is the hash index that contains the largest number of attributes in sr_1 and 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 $A3$. 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 creating a new hash index on $A3$. But this would henceforth add additional memory and maintenance costs for each stored tuple.

To support such diversity of criteria requires multiple 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 heavy update loads experienced by DSMS. The inefficiency of deploying multiple hash indices over a state is confirmed by our experimental study both on synthetic and on real data sets (Section V).

B. AMR Index Requirements

Traditional off-line index tuning approaches that select the "best" fixed index structure off-line [6], [7], [8] 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 [9], [10], [11]. Clearly, AMR systems require an online index tuning approach.

AMR systems have unique challenges. 1) Periodically the router sends search requests to suboptimal operators to update system statistics [3]. The extremely low frequencies of these suboptimal search requests are not likely to influence the final indices selected, yet they add additional overhead. Thus keeping too few statistics may reduce overhead but at the expense of not being able to locate the most optimal indices. While keeping detailed statistics may be able to accurately locate the "best" indices, but require higher overhead. 2) To online tune indices requires the system to continuously assess and adjust indices to the criteria of the query paths of the search requests in the system. The abruptness and frequency of changes in the query paths in AMR systems makes this extremely challenging. Furthermore the overhead of assessing indices clearly must not detract from producing rapid results.

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

C. 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, and 3) compact enough to fit in main memory. In addition, AMR systems require an index tuning approach that reduces required system resources while maintaining the integrity of the indices selected. 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 contributions include:

1) We propose an index design that serves workloads composed of multiple access patterns while remaining efficiently maintainable in highly dynamic DSMS.
2) We develop two index assessment methods, *Compact Self Reliant Index Assessment CSRIA* and *Compact Dependent Index Assessment CDIA*, that reduce the required system resources by eliminating statistics. *CSRIA* utilizes a heavy hitter method [12] 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 [13] to compact the statistics.
3) Our extensive experimental study demonstrates that our *AMRI* solution always wins over the current AMR indexing methods, including traditional hash indices [5] and bitmap indexing [14]. For diverse synthetic stream data AMRI produced on average 93% more results (cumulative throughput) than the current indexing approach and 75% more results than the bitmap indexing approach over the same period of time.

The paper is organized as follows. Section 2 defines terms. Section 3 describes AMRI's physical design. Section 4 presents assessment methods. Sections 5, 6, and 7 cover experimental analysis, related work, and conclusion.

II. Preliminaries

We consider SPJ (select-project-join) queries using standard sliding window semantics (Figure 2). The solution is explained using a single SPJ query. However, our proposed logic equally applies to 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 (i.e., a state for each of the following: StreamA, StreamB, and StreamC). 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.

A *join predicate* is expressed in the WHERE clause of a query composed of 1) a join expression ($=$, $<$, $>$, $\geq$, $\leq$), 2) an attribute stream reference *S1.a1*, and 3) another attribute stream reference*S2.a2* (e.g., $A.A1 = B.A2$). The *join attribute set*, or *JAS*, for a state is the set of 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 based on any combination of attributes in *JAS*. An *access pattern* (ap) is a combination of *JAS* attributes 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 or ja used to search is represented by a capital letter naming the attribute, while a ja not used to search is represented by the special wild card symbol $*$.

For example consider the state of Stream A with jas $A1$ and $A3$. $< A1, A3 >$ is an ap indicating that all join attributes are used to search on. While $< A1, * >$ is an ap only using ja $A1$, and $< *, * >$ is an ap using no join attributes (i.e., a full scan).

III. AMRI Physical Design

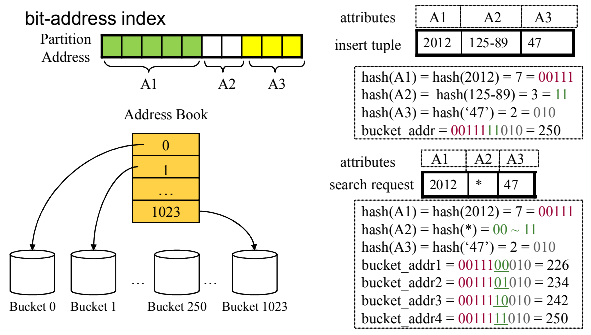
Instead of maintaining multiple distinct hash indices we propose to employ a versatile yet compact bit-address index as the foundation of our solution [14]. We now explain how this meets the AMR requirements (Section I-B).

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 created 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 the mapping. The IC derived for each tuple is never stored.

Reconsider the example from section I-A. Now ponder inserting tuple t, a *package* record, into the bit-address index BI solution in Figure 3 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 [5] (Section I-A) which utilizes both memory and CPU time to create and store 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 BI_1 to BI_2 requires the relocation of each 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 3. 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

Now consider search request sr_1 looking for all packages with priority code $=$ 2012 and location id = 47 using the bit-address index solution in Figure 3. 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 special wild card symbol $*$ or in the case of sr_1, $t.A2$ whose bit values cover 00 to 11). Then the values are combined into bucket ids (i.e., 0011100010, 0011101010,0011110010, and 0011111010). Finally a scan is performed across buckets 226, 234, 242, and 250. If the search is narrow and precise (i.e., the access pattern specifies all join attributes) then a single bucket will need to be searched. If the search is wide (i.e., the access pattern contains wild cards) then several buckets will need to be searched. There is no clear winner between searching using a BI versus using multiple hash keys. On average we expect a good index configuration to limit the number of buckets 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.

The configuration of the index key map influences the number of buckets evaluated during the search. The optimal index key map is configured so that no bucket stores more tuples than any other bucket (i.e., a even distribution of stored tuples). Index key map selection is a generic hashing issue and not specific to inherent challenges of AMR systems. To simplify the presentation, we assume that the range and estimated distribution of each attribute is known.

IV. Index Assessment

The index *assessment* component maintains compact statistics used to locate the index configuration with the lowest index configuration dependent costs C_D.

A. Index Configuration Dependent Costs C_D

The quality of an index configuration IC depends upon the estimated total *unit processing cost* for IC [15], or the combined maintenance and search costs [14]. The maintenance cost is the sum of the cost to insert and delete tuples in a state and to compute bucket ids ($C_{insert} + C_{delete} + C_{hash,I}$). The search cost is the sum of the cost 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) [14]. See notations in Table I [14].

Table I
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

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

B. Access Pattern Statistics

To assess possible index configurations, statistics must be collected 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 each state tracks the number 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 $\sum_{k=1}^{N_{ja}} \binom{N_{ja}}{k}$ which is exponential in the number of join attributes. We now outline our index assessment approaches.

C. Self Reliant Index Assessment SRIA

1) Basic SRIA: The naive index assessment algorithm, called *Self Reliant Index Assessment* captures access pattern statistics 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 enable quick access to each access pattern ap in the *SRIA* table, we create direct referencing by mapping each ap to a unique binary representation $BR(ap)$. 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 $BR(ap_1) = 100$ which represents ap number 4. But if ap_1 is searching using attributes B and C (i.e., $ap_1 =< *, B, C >$) then $BR(ap_1) = 011$ which represents ap number 3.

Each incoming access pattern ap is processed using the $BR(ap)$ 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.

2) Compact SRIA: Access Pattern Reduction: The large number of possible access patterns (Section IV-B) can cause memory limitations. We now explore an access pattern reduction method extension to *SRIA* modeled after the heavy hitter algorithm proposed by Manku et. al. [12] referred to as *Compact SRIA* or *CSRIA*. During assessment *CSRIA* periodically removes any access pattern statistic whose frequency falls below a preset error rate or ϵ. At the end 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 evaluates incoming access patterns in segments. Each segment contains $\lceil \frac{1}{\epsilon} \rceil$ search requests. Segments are associated with an id that represents the 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 thus far. To ensure that access patterns are not deleted too early, each time an ap is added to the SRIA table the current maximum error in frequency δ along with it. δ represents the minimum number of requests that should have occurred in order for f_{ap} to be above s_{id}.

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. During assessment, creating statistics from the access patterns of incoming search requests (*insertion*) proceeds as follows: First, the binary representation $BR(ap)$ 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., $f_{ap} + \delta \geq \theta - \epsilon$.

There are several benefits to this approach: 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 II
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 different access patterns. This decreases an opportunity to find the optimal IC. Consider a state with 3 join attributes, SRIA Table II, and a 4 bit IC. If θ is 5% and ϵ is .1%, then *CSRIA* will delete the access patterns $< A, *, * >$ and $< A, B, * >$ even though both access patterns would benefit from an IC containing $< A, *, * >$. Furthermore the combined frequency of $< A, *, * >$ and $< A, B, * >$ is 8% which is greater than θ. The IC found by *CSRIA* is the configuration with the B attribute having 1 bit and the C attribute having 3 bits. Whereas the true optimal IC is the configuration with A and B attributes having 1 bit each and the C attribute having 2 bits.

D. Dependent Index Assessment DIA

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 IC of a state. Then we describe how the dependent access pattern relationships can be used in a 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*, the IC consists of no attributes in ap (e.g., IC consists of attributes C and D). IC and ap have no common attributes. No buckets can be eliminated. A comparison of all tuples stored in the state is required.

In a *slightly better case*, IC consists of some attributes in ap and some 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 III. 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 that consists of attributes A and B). Since all attributes in IC are specified in ap, one single bucket will be searched. Further, all tuples in the bucket will correspond to matches. 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.

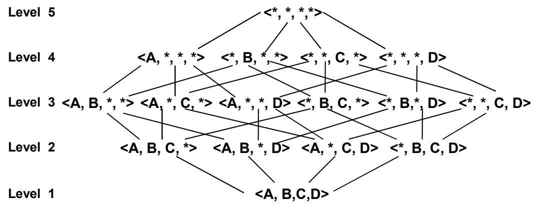
Definition 1: An index based upon access pattern ap_1 provides a **search benefit** to a search request utilizing access pattern ap_2, denoted as $ap_1 \prec ap_2, if \forall a_i \in ap_1 \Longrightarrow a_i \in ap_2$ where a_i is an attribute in ap_1.

The search benefit relationship organizes the access patterns into a lattice (Figure 4). Each node in the lattice corresponds to an ap. The lattice is formed from a single node representing the ap that contains no join attributes (top). 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 all possible join attributes are included in a single node (bottom). 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 4.

Dependent Index Assessment DIA stores the assessment values in a lattice to retain the dependent search benefit relationships between access patterns. 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 access pattern, if a node exists in the lattice that matches it then the corresponding count $N.A_{ap}$ is

Figure 4. State with 4 join attributes



incremented by 1. Otherwise, a new node is created for the ap. To enable quick direct referencing to ap, 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* nodes is stored in a *SRIA* table.

2) Compact DIA: Access Pattern Compression: The large number of possible access patterns in DIA (Section IV-B) can cause memory limitations. We now explore an access pattern reduction method extension to *DIA* modeled after an elegant hierarchical heavy hitter algorithm proposed by Cormode et. al. [13] 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* $(\forall k \in P_{AP}\, (k \notin Q)\, and\, (ap \prec k))$.

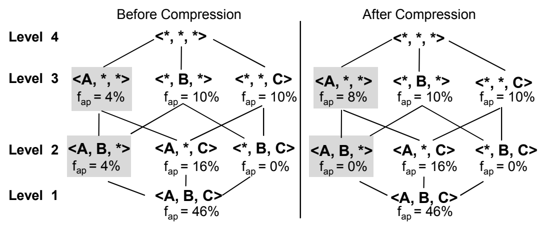
The *CDIA* approach is composed of three methods *insertion* (creating statistics), *compression* (combining statistics), and *final results* (finding all statistics that meet the threshold). *Insertion* works by evaluating incoming access pattern in segments in the same fashion as outlined for CSRIA. 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 located 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 combination strategies can be utilized [17]. One method, *random combination*, randomly picks a parent node. Another method, *highest count combination*, adds the child's frequency to the parent with the largest 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.

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 [13]. 3) It reduces the number of access patterns stored while retaining the statistics of removed access patterns.

Figure 5. CDIA Example



Reconsider the example in Section IV-C2 with a state containing 3 join attributes, a 4 bit index configuration, and DIA outlined in Figure 5. 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, CDIA will find the true optimal index configuration.

V. Experimental Results

Experimental Setup: Experiments are implemented in the CAPE stream system [18] using Linux machines with AMD 2.6GHz Dual Core CPUs and 4GB memory. They measure throughput defined as the cumulative output tuples produced. Our experiments explore: 1) Which of our proposed assessment methods is the most effective at improving the AMR throughput? and 2) Is AMRI more effective at improving throughput than state-of-art AMR indexing?

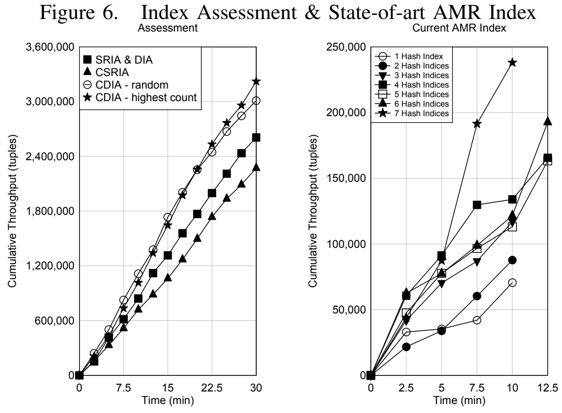
Synthetic Data Sets: To test the effectiveness of our methods under conditions where adaptive indices are required we created synthetic data in which the selectivities of joining one stream to another adapt over time. This may cause the router to use new query paths which in turn may initiate the selection of new indices.

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 (i.e., 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 aps, there already is a significant benefit in removing a single ap. Clearly, as the number of ap in a state increases so does the probability of ap statistics being eliminated.

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 aps by running the quasi training data approach above.

Index Assessment: First, we compare index assessments methods SRIA, CSRIA, DIA, and CDIA using both random and highest count compression. Each method is run using the maximum error $\delta = .05$ and threshold $\theta = .1$. Both CDIA versions outperform DIA, SRIA, and CSRIA (Figure 6). 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) . Note that DIA's and SRIA's results are equal, because both approaches share the same code base, use the same SRIA table, and do not reduce any nodes.

Figure 6. Index Assessment & State-of-art AMR Index

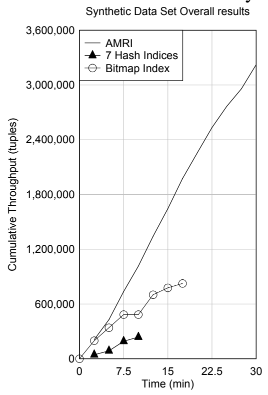


State-of-art AMR Index Approach: Next, we evaluate the state-of-art AMR indexing [5] using highest count compression CDIA outlined above. The synthetic data set and query were run with the minimum (1) to maximum (7) number of possible hash indices. Static non-adapting hash indices (i.e., no index tuning) produced poor results. Thus adaptive hash indices that utilize highest count compression CDIA index tuning and conventional index selection (i.e., indices created support the most frequent search request access patterns) are used.

None of the trials ran for more than 12.5 minutes (Figure 6). 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 produces 93% more results than even the best hash index configuration (Figure 7).

Index Tuning: We now compare the AMRI index tuning to a non-adapting 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 7). Additional concepts and experiments using real data can be found in [19].

Figure 7. State-of-art vs AMRI in Synthetic Data



VI. Related Work

The first AMR system, Eddy, used a router to route individual tuples through operators [3]. Enhancements in routing policies were proposed in [4], [20]. Multi-query AMR system issues were covered in [21]. [5] improved optimization capabilities by extending routing flexibility through the development of the STeM operator and access modules. As shown in our experiments the overhead of access modules makes such hash indices ineffective.

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 [6], [7], [8] or online [9], [10], [11]. Online index tuning continuously evaluate and adjust the index configuration to the current workload. Our solution borrows from the online index tuning work. We explore novel index tuning approaches that reduce the system resources required while maintaining the integrity of the index selection evaluation in accordance with a preset threshold and error rate.

Bit-address indexing, initially designed to index partial match queries on a file [22], has been applied for a variety of applications ranging from compactly storing very large multidimensional arrays [23] to reducing processing costs of multidimensional queries [24]. [14] studied index selection heuristics using a bit-address index where the search request workload is known prior to execution. We instead tackle the on-line index tuning problem for AMR systems.

Assessment methods for bit-address index have not been studied while they are core to our effort. Such methods utilize stream sampling algorithms. Heavy hitter algorithms, a type of stream sampling algorithm, meet the requirements of AMR systems (Section I-B) as they analyze and report all items that appear above a preset threshold [25], [12].

[25] introduced the first deterministic algorithm for approximating frequency counts, called the heavy hitter method. [12] added the error rate ϵ approximation guarantee. Our *CSRIA* method is modeled after the heavy hitter algorithm proposed in [12]. Hierarchical heavy hitter, applying the heavy hitter methodology to hierarchical multi-dimensional data, was studied in [13]. Our *CDIA* method is modeled after this hierarchical heavy hitter work. CDIA implements two compression methods customized to handle the search benefit relationships between indices in AMR systems, namely, *random combination* and *highest count combination*. Similar compression strategies were presented by [17]. To our knowledge, ours is the first application of heavy hitter and hierarchical heavy hitter algorithms to address the problem of index tuning in general and tuning in AMR systems in particular.

VII. Conclusions

We developed the Adaptive Multi-Route Index for AMR systems. We propose 4 customized AMR systems assessment methods (SRIA, CSRIA, DIA, and CDIA). Our experiments demonstrate the overall effectiveness of our AMRI at improving throughput in dynamic stream environments. In particular, using synthetic data AMRI produced on average 93% more results than the state-of-art approach.

VIII. 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. Abadi and et. al., "Aurora: a new model and architecture for data stream management," *VLDB*, pp. 120–139, 2003.

[2] T. Urhan and et. al., "Cost-based query scrambling for initial delays," *SIGMOD*, pp. 130–141, 1998.

[3] R. Avnur and et. al., "Eddies: continuously adaptive query processing," *SIGMOD*, pp. 261–272, 2000.

[4] P. Bizarro and et. al., "Content-based routing: different plans for different data," in *VLDB*, 2005, pp. 757–768.

[5] V. Raman and et. al., "Using state modules for adaptive query processing," *ICDE*, p. 353, 2003.

[6] S. Agrawal and et. al., "Automated selection of materialized views and indexes in sql databases," in *VLDB*, 2000, pp. 496–505.

[7] ——, "Database tuning advisor for microsoft sql server 2005," in *SIGMOD*, 2005, pp. 930–932.

[8] B. Dageville and et. al., "Automatic sql tuning in oracle 10g," in *VLDB*, 2004, pp. 1098–1109.

[9] S. Agrawal and et. al., "Automatic physical design tuning: workload as a sequence," in *SIGMOD*, 2006, pp. 683–694.

[10] K. Schnaitter and et. al., "Colt: continuous on-line tuning," in *SIGMOD*, 2006, pp. 793–795.

[11] N. Bruno and et. al., "An online approach to physical design tuning," *ICDE*, pp. 826–835, 2007.

[12] G. Manku and et. al., "Approximate frequency counts over data streams," in *VLDB*, 2002, pp. 346–357.

[13] G. Cormode and et. al., "Finding hierarchical heavy hitters in data streams," in *VLDB*, 2003, pp. 464–475.

[14] L. Ding and et. al., "Index tuning for parameterized streaming groupby queries," in *SSPS*. ACM, 2008, pp. 68–78.

[15] J. Kang and et. al., "Evaluating window joins over unbounded streams," *ICDE*, p. 341, 2003.

[16] T. Johnson and et. al., "Sampling algorithms in a stream operator," in *SIGMOD*, 2005, pp. 1–12.

[17] G. Cormode and et. al., "Diamond in the rough: finding hierarchical heavy hitters in multi-dimensional data," in *SIGMOD*, 2004, pp. 155–166.

[18] E. A. Rundensteiner and et. al., "CAPE: Continuous query engine with heterogeneous-grained adaptivity," in *VLDB*, 2004, pp. 1353–1356.

[19] K. Works, E. A. Rundensteiner, and E. Agu, "Index tuning for adaptive multi-route data stream systems," Worcester Polytechnic Institute, Technical Report WPI-CS-TR-10-05, 2010.

[20] F. Tian and et. al., "Tuple routing strategies for distributed eddies," in *VLDB*, 2003, pp. 333–344.

[21] S. Madden and et. al., "Continuously adaptive continuous queries over streams," in *SIGMOD*, 2002, pp. 49–60.

[22] A. Aho and et. al., "Optimal partial-match retrieval when fields are independently specified," *ACM TDS*, pp. 168–179, 1979.

[23] D. Rotem and et. al., "Chunking of large multidimensional arrays," Lawrence Berkeley National Lab, Tech. Rep. LBNL–63230, Feb 2007.

[24] ——, "Towards optimal multi-dimensional query processing with bitmapindices," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL–58755, Sep 2005.

[25] J. Misra and et. al., "Finding repeated elements," Cornell University, Ithaca, NY, USA, Tech. Rep., 1982.