Scalable ordered indexing of streaming data

Sobhan Badiozamany
Department of Information Technology
Uppsala University
Box 337, SE-751 05,
Uppsala, Sweden
Sobhan.Badiozamany@it.uu.se

Tore Risch
Department of Information Technology
Uppsala University
Box 337, SE-751 05,
Uppsala, Sweden
Tore.Risch@it.uu.se

ABSTRACT

In order to efficiently answer continuous queries requiring range search in large stream windows, data stream management systems (DSMSs) need ordered indexes. Conventional DBMS indexing methods are not specifically designed for data streaming applications with extremely high insert and delete rates for windows over streams. This motivates a scalability investigation for various ordered main memory indexing methods in a streaming environment, through implementation and experiments. Our experimental studies show that a state-of-the-art implementation of cache-aware compact tries is a very suitable indexing structure for data streaming applications allowing constant time insert and access rates. However, in the best of the investigated implementation the range search was slow. Since a highly optimized implementation of compact tries is very complex we developed a framework for scalable range search in an index without any change to its source code. Another important issue is that index maintenance in window based data stream environments require a scalable way of deleting data, which is addressed by an index independent window aware bulk deletion technique, also without changing any source code.

1. INTRODUCTION

A Data Stream Management System (DSMS) usually has a local main memory database against which high volume streaming data is matched. This local database includes storage for windows of data streams flowing through the system. The windows may become large, so indexing data in stream windows is an interesting problem. In many cases ordered indexing is needed, which is investigated here.

The requirements of data stream indexing is not exactly the same as conventional DBMS indexing, causing some traditional DBMS indexing structures to fall behind the requirements of DSMS applications. The following are important differences between conventional DBMS indexing and DSMS window indexing:

- Because of very high stream rates DSMS indexes need to be stored in main memory and the indexing data structure should be main-memory oriented, i.e. be CPU cache conscious and compact.
- Stream window indexes need to be able to handle very high insert, update, and delete rates. By contrast, most conventional DBMS applications behave based on a high watermark. That is, once the database is filled up, it does not rapidly grow or shrink in size. In other words, DBMS applications have lower demand for massive insertion and deletion than DSMS application while fast search is desirable in both.

In this paper we investigate the performance of different kinds of ordered indexing methods for main memory databases in context of window based stream processing w.r.t. the three aspects of ordered indexing for massive data streams: insertion, search, and deletion. The goal is to find the best suitable ordered sliding window indexing method for massive data streams.

To improve the performance of deletion from indexes over time stamped stream windows, we propose a window aware indexing maintenance method, *partitioned temporal window index (PTWI)*, and through experiments we show that it outperforms a naïve incremental index deletion strategy. In addition PTWI can be used together with any kind of underlying indexing structure.

We implemented and compared the performance of indexing sliding windows over data streams of main memory B-trees, cache sensitive B+ trees [12], burst tries [10], and the highly optimized but complex compact trie implementation Judy [18]. For empirical investigations we used randomly generated synthetic data as well as data generated by the Linear Road Benchmark (LRB) [1] for streaming data. Benchmark queries were used to compare the scalability of insertion, deletion, and range search for different indexing structures.

Judy is a highly optimized compact trie implementation that focuses on both compactness and CPU cache utilization to improve the performance. However, the current Judy implementation lacks efficient range search iteration, as also noted by [15] [16]. To improve memory and CPU cache utilization, Judy dynamically changes between its around 50 different internal node structures based on the current key distribution in each node, which makes the implementation of Judy very complex and difficult to change. We therefore developed a method to improve range search in a complex index implementation such as Judy, without changing its source code. The experimental results show that our extended version of Judy scales the best among the other investigated main memory indexing structures.

This paper is organized as follows. Section two makes an overview of related main-memory ordered indexing methods. Section three first defines the benchmark scenario and then describes required extensions to the indexing methods for range search and massive deletion. Section four evaluates the scalability of the different indexing methods through experiments on implementations. Section five summarizes the result and proposes future work.

2. Background and related work

Sampling techniques like *window aware load shedding* [19] have been proposed for processing approximate queries when the stream rates are higher than the DSMS can handle. Load shedding is not suitable when all stream elements in the window must be maintained, such as in monitoring communication networks [2] and urban traffic [1].

A complement to load shedding is indexing. Proper indexing increases the performance of the DSMS and decreases the need for sampling techniques. We have investigated the performance of the most common main memory ordered indexing structure for our setting. In particular we review different kinds of B-trees and tries.

The compact trie implementation Judy was found to be particularly interesting to investigate. However, Judy needs some extensions for supporting efficient streaming range search and massive deletion. Since the implementation of a highly optimized compact index structure such as Judy is very complex, we have devised methods to improve range search and deletion for an index implementation without altering it.

B-trees

The CSB+ variants of B-trees [12][9] and the binary T-tree [14] have been proposed to index main memory data in a cache conscious way. A recent study [12] suggests that in the context of in-main-memory indexing on modern processors T-trees do not perform better than classical B-trees. Therefore classical B-trees regained the research focus and there has been attempts to make B-trees cache conscious [2] [3]. By exploiting the CPU cache more effectively, the CSB+ tree improves the search time at the cost of using more space and slightly slower insertion and updates than regular B+ trees [2]. We show that the major problem with CSB+ trees compared to B-trees is space inefficiency.

Tries

In the simplest form, a trie is a multi-way tree structure in which each node is an array of pointers. The size of each array is equal to the number of letters in the alphabet, e.g. 26, and each level in a trie indexes a letter in a word. The main advantage of tries is constant insertion and access time if the length of the key is fixed. Thus tries should be very well suited for indexing data stream windows with very high insert rates. Figure 1 shows a naïve trie. Each node in the trie represents a *sub-expanse* [18], which is a set of keys that are accessed through it. In Figure 1, all keys in the range [COAAA,COZZZ] are in the same sub-expanse accessed through the node marked as "CO".

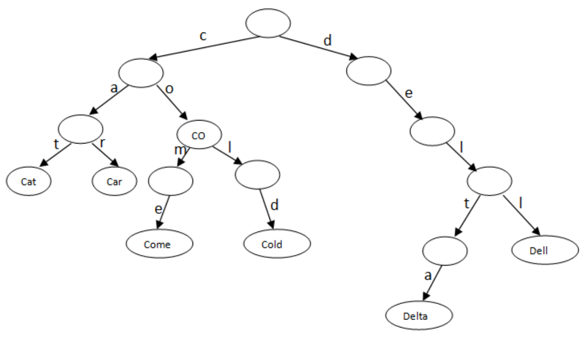

Figure 1. A naïve trie example that stores string keys "cat", "car", "cone", "cold", "dell", and "delta".

Although tries were originally introduced to index character strings, they can be easily modified to index any ordered domain. An order preserving key transformation function can be defined that returns a binary key representing the rank of the original key in the domain. If prior knowledge about the domain exists, such a transformation can be done on-the-fly as done by, e.g., [12]. A binary key can then be indexed by breaking it down into bytes and then introducing them to the trie like characters of a string. For simplicity, here we consider the binary keys to be 32 bit integers broken into 4 bytes. In a naïve implementation for integer keys, the trie is then always 4 levels deep. Each node is a simple array of 256 pointers to the nodes in the next level or, in case of nodes in the 4th leaf level, pointers to values. Tries can be extended to support longer integers and other forms of breaking integers [6].

The memory utilization problem with tries is that they are sensitive to the distribution of keys. In the worst case, when the keys are uniformly scattered across the whole domain, naïve tries waste memory because there will be many null pointers in the sparse pointer arrays representing trie nodes. Several compression techniques have been introduced to overcome naïve tries' weak memory utilization [5] [10] [13] [18]. The main objective in most of them is to achieve a compact representation that, despite its compactness, can still support constant insertion/search time.

A burst trie [10] is based on the idea that as long as the population is low, keys that share the same suffix can be stored in the same *container*. Containers are sorted lists of partial keys together with their associated values. During index lookup, once the right container is found, the key is located using binary search. Containers have a limited capacity and therefore, in an attempt to insert more keys into a full container depending on the implementation particulars, the container is transformed into a larger internal node, and thus 'bursts' into several new containers. The keys will thereby be redistributed to the new containers based on deeper suffix calculations, and the pointers in the new internal node will refer to new containers. This is an effective approach to decrease memory consumption. However, since the container capacity is fixed in all nodes, the internal nodes often still have null pointers and the memory utilization can still be a problem.

Judy compact trie implementation

Judy [18][3] can be categorized as a variation of burst tries, but with an important distinction: the node (container) data structure and its size is not fixed. To improve memory and CPU cache utilization, Judy dynamically changes node structures according to the current distribution of keys in each node choosing among around 50 different representations of internal nodes. Judy is a highly tuned but very complex data structure. Judy's approach towards an efficient compression technique is to use a variety of compact node structures that fit in a single cache block for different kinds of local sub expanse populations. This allows the contents of any kind of node to be moved to the CPU cache for fast consecutive access. Furthermore, Judy maintains the most interesting characteristic of tries. That is, the depth of Judy is constant, e.g. for indexing integer keys Judy is always 4 levels deep. This means constant time is guaranteed for all single element operations.

Judy supports iteration based range search. However, in the current Judy implementation the iterator always starts at the root, which makes it perform worst among the investigated methods w.r.t. range search. The J+ tree [16] and PJ+ tree [15] address this problem by introducing a sorted linked list as an extra level of leaf nodes. We were unable to obtain the source code for J+ or PJ+ trees for making an empirical evaluation. However, compared to Judy, the J+ tree worsens the performance of single key operations in Judy because it adds an extra level of search and maintenance of the leaf node lists; it also consumes much more memory since prefixes are stored uncompressed in the leaf node lists. The prefetching variant of the J+tree, the PJ+ trees [15], improves the range search performance by adding prefetching pointers, but does not address any of the J+ tree deficiencies.

Non intrusive range search

Implementations of indexing structures for highly tuned indexing structures such as Judy might become very complicated. To improve software reusability and eliminate unnecessary modifications to highly optimized implementations, we use *mappers* as a general method that simplifies traversal of data structures. A mapper is a second order function that applies a *mapping function* on a set of elements. In the ordered indexing context a mapper is a function that traverses a range of keys specified by low and high bounds, and applies a user provided mapping function on the key-value pairs in the range.

Using mappers we added range search to Judy without any modification to it. We show that the mapper approach substantially improves range search compared to the built-in implementation. This makes Judy extended with mappers perform better than other investigated approaches.

GIST [11] is a general framework for adding tree-based indexes to an extensible DBMS for supporting range search. It is challenging to make the code changes required by GIST for a complex trie structure such as Judy, and we therefore instead used mappers.

Non intrusive bulk deletion for sliding windows

If there is massive stream flow through a tumbling window, deleting the expired stream elements from the window index becomes an issue. Naive element by element deletion is slow. Common methods to speed up bulk insertion and deletion are to use partitioned indexes and create/delete entire partitions in bulk [17] or prefixing keys in a B-tree with partition identifiers [7]. We adapted partitioned bulk deletion to support non-intrusive bulk deletion of indexes over sliding time stamped windows, called *partitioned temporal window index* (PTWI). The main difference to regular bulk deletion is that PTWI maintains a circular array of pointers to time stamped sub-window indexes, which are completely deleted as the main window slides.

3. Ordered indexing of data streams

We address three main challenges in indexing data in sliding windows: scalable insert, fast range search, and scalable deletion. The suitability of several indexing methods w.r.t. these aspects have been investigated. For the investigation of the methods we used own implementations, publically available implementations, and publicly available versions extended with our improvements.

3.1 Scenario

To analyze the problems of maintaining proper ordered indexing structures for window based stream processing and comparing scalability of different indexing solutions, we use the Linear Road Benchmark data generator. It generates for a predefined number of expressways *L* an input data stream with the following tuples:

[T, X, D, S, VID, VEL]

Where

- T is a time stamp.
- X is the expressway on which a vehicle is traveling, 0 to *L*-1.
- D is the direction in which the vehicle is traveling, which is either east or west.
- S is the segment of the expressway, 0 to 99.
- VID is the vehicle's identifier.
- VEL is the speed of the vehicle.

Our performance evaluation simulates index search for the following index intensive queries:

- Q1: What is the velocity of a specific vehicle *v* on expressway *x* traveling in direction *d* in segment *s* during the last minute? This query selects a single tuple.

```
select VEL
from [last minute window]
where X=x and D=d and S=s and VID =v;
```

- Q2: What is the average velocity of all vehicles on expressway *x* traveling in direction *d* in segment *s* during the last 5 minutes? This query is selecting *1/L %* of the position reports in the window.

```
select average (VEL)
from [last 5 minutes window]
where X=x and D=d and S=s;
```

An ordered index on the compound key <*X, D, S, VID*> provides scalable answers to both queries. The VID attribute needs to be included in the ordered index since, at traffic peaks, LRB generates a large number of vehicles per segment in a minute (around 100,000).

Query Q1 accesses a single element in the index having the key <*x,d,s,v*>.

Query Q2 is a range search where the lower limit of the compound key is <*x,d,s,0*> and the upper <*x,s,d,∞*>.

Since the main window covers 5 minutes and tumbles every 1 minute, the older data on the index must be removed, which requires massive deletion from the index.

3.2 Improving range search on Judy

The most common way to iterate over index ranges is to use a Volcano style scan structure with a *next* method [8]. Such a structure is indeed available in Judy, but it does not perform well because the *next* method always starts from the root in the current implementation, without using a scan data structure. For scalable range search, Judy has to be modified. However, to implement a scan data structure in a highly complex indexing structure such as Judy, having over 50 different node types, is a challenging task since all state information has to be continuously maintained in the scan. The alternative to implement scans using linked leaf nodes as in B+ trees would require substantial modifications of Judy with unknown consequences.

To add efficient range iteration to Judy without the complexity of implementing scans, we instead implemented a second order C *mapper* function that applies another C function on every key-value pair in a given key range. This approach requires no change to Judy and no explicit code to maintain the complex state information as in scans. Our implementation also supports generic iteration over scans by using threads combined with a buffer of recently mapped key-value pairs.

Listing 1 shows the general signatures for mapper and mapping functions in C for range search operations in an ordered indexing structure.

Listing 1, general mapper and mapping functions for range search

```
typedef int (*mapping) (key k, value v,
                        void *xa);
Mapper(indexroot* tree, key lower, key,
       upper, mapping m,void *xa);
int SumMapping(key* k,value* v,void *xa)
{
  *(int *) xa += (int) *v;
  return TRUE;
}
```

The following code traverses the index structure pointed to by `tree` in the range [100,200] and applies `SumMapper` to all key-value pairs in the range. The sum of the values are accumulated in the variable *sum* passed by reference to the mapper.

```
key k1=100; k2=200;
int sum=0;
indexroot* tree=new_index();
Mapper(tree, k1, k2, SumMapping, &sum);
```

Based on the general mapper paradigm, we implemented a mapper function for Judy that performs the range search. The mapper recursively visits the nodes that cover sub expanses which are within the specified range. For each leaf node, it applies the mapping function to the key-value pairs in the leaf nodes that are within the range. In Judy the bytes of the key are not always implicitly stored in nodes, so the algorithm has to carry a prefix at any call level. Listing 2 provides an outline of the algorithm (the C code can be downloaded from [21]).

Listing 2, Judy mapper

```
JudyMapper(Judypointer jp, key lower, key
upper,  key prefix, mapping fn,void *xa)
{
  switch (type (jp) )
  {
    case internal_nodes:/* many variants
         of linear,bitmap,uncompressed */
      for all Judy pointers p in each
              internal node that covers
              the range [lower, upper]do
      {
        Update the prefix;
        JudyMapper(p, lower, upper,
                   prefix, fn, void *xa);
      }
    case leaf_nodes:/* linear, bitmap or
                       immediate leafs */
      for all keys k inside range
              [lower, upper]do
      {
        Construct the key by extending
           prefix;
        Find value v associated with k;
         (*fn)(k,v,xa); /* apply mapping
                           function */
      }
  }
}
```

3.3 Window aware index deletion

We compare two different strategies for deleting time stamped elements from indexes over sliding windows: naive incremental deletion and the bulk window index deletion method PTWI.

3.3.1 Incremental deletion

In incremental deletion there is only one indexing structure for the whole window. In order to identify the right set of keys to be deleted, the time stamp has to be explicitly stored as a part of the key. The index key thus takes the form of <*t, k*> where *k* is the application key (i.e. <*X, D, S, VID*> in LRB) and *t* is the time stamp associated with it. Notice that the order in the compound key proposed here preserves the temporal order of keys. Therefore, deletion is straight forward; after the time stamp *t* expires, all keys of form <*t,* *> need to be removed. Since an ordered indexing structure is used, all keys in this range are found and then deleted from the index one by one.

Naive incremental deletion of keys one by one might take considerable amount of time since the data structure is searched from the root for each deleted key.

3.3.2 Bulk window index deletion

As an alternative to incremental deletion we also implemented a special bulk deletion technique for sliding time stamped windows called *partitioned temporal window index* (PTWI).

PTWI is applicable for sliding windows. Let *N* be the time span of the window and *S* be the stride for the sliding in time units. At each slide a sub-window of size *M*=*N/S* tumbles. In LRB *N*=300 seconds and *S*=60 seconds, thus *M*=5. With PTWI *M* nonoverlapping partial indexes are maintained for the whole sliding window. When the window slides, the partial index that stores the oldest subwindow is dropped and a new empty partial index is created. PTWI is implemented as a one dimensional circular index array of size *M* of pointers to partial indexes, as illustrated by Figure 2.

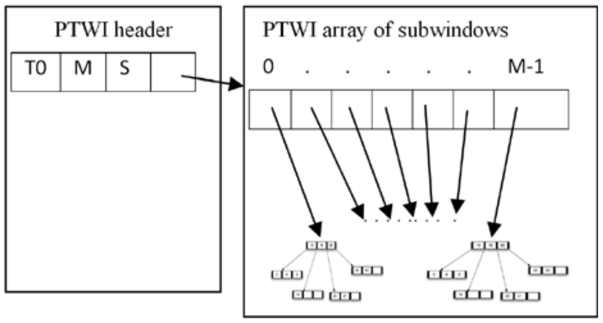

Figure 2. The PTWI structure

In the *PTWI header* the following information is maintained as the window slides:

T0: Starting time for the indexed stream, initialized to the time for the first arriving tuple.

M: Number of subwindows. In LRB *M*=5.

S: The stride of the subwindows as time units. In LRB *S*=60 seconds.

When a new tuple with time stamp *t* and data tuple *tpl*, <*t, tpl*>, arrives in the stream, the system first determines whether tumbling of a subwindow is needed or not. Tumbling is needed when *mod(t,S)=0*.

a) If *mod(t,S)≠0*, i.e. no tumbling, the system computes the position *i* in the subwindows array containing a pointer to the subwindow index where *tpl* should be inserted, accessed, or updated:

   *i = mod(t-T0,W*S)*

b) When *mod(t,S)=0* the oldest window tumbles by completely dropping it from the subwindow array and replacing it with a new empty window index. The position *d* in the subwindows array for the window index to replace is computed by:

   d = mod(t/S,W)

For example, Figure 3 illustrates the evolution of the subwindows array for the LRB scenario. In the beginning of any minute *T*, the oldest partial index associated to minute *T*-5 needs to be dropped and a new empty one for minute *T* is created. Figure 3 shows the content of the subwindows array during minutes 1 to 10. In each minute *T* incoming data is inserted only to the index associated with the current minute, tagged as @*T* in the figure.

Queries that access a single tuple at a given time point *t*, such as Q1, can be directly answered by calculating *i* as in a) and then accessing window index *i* in the PTWI array of subwindows.

To answer queries that cover the whole window, they have to be divided into sub-queries – one for each minute – and their results merged. For example, for query Q2 the time period is the last 5 minutes and therefore the range query [<*x,d,s,0*>, <*x,s,d,∞*>] for given *x, s,* and *d* is issued over all 5 subwindow indexes in the array.

Notice that bulk deletion can be done in a lazy manner in a background process. In other words, deletion is no longer a burden on the real time expectations of the system.

Furthermore, notice that any kind of indexing structure can be used for storing the subwindow indexes.

The space overhead of PTWI compared to incremental deletion is negligible, since it just adds one extra array of *M* pointers and the PTWI header. The computational overhead is one extra simple numerical computation per stream tuple to obtain *i*, while *d* is computed only when the window tumbles.

Figure 3. Contents of the *PTWI* array of subwindows during first 10 minutes, with 5 minutes window size and 1 minute stride. @*T* represents the pointer to the subwindow index for minute *T*.

The PTWI's window index array					minute
@1	nil	nil	nil	nil	1
@1	@2	nil	nil	nil	2
@1	@2	@3	nil	nil	3
@1	@2	@3	@4	nil	4
@1	@2	@3	@4	@5	5
@6	@2	@3	@4	@5	6
@6	@7	@3	@4	@5	7
@6	@7	@8	@4	@5	8
@6	@7	@8	@9	@5	9
@6	@7	@8	@9	@10	10

4. Experimental evaluation

We experimentally compared the scalability of insertion, single element retrieval, incremental deletion, bulk deletion, and range search in a B-tree, CSB+tree, Burst trie, Judy, and Judy extended with efficient range search. We also compared the performance of PTWI with incremental deletion for Judy and B-trees. The outcome supports the initial hypothesis that Judy extended with efficient range search and PTWI outperforms other investigated in-main-memory indexing structures. The succeeding sections describe how tests were performed and present experimental results.

4.1 Experimental setup

The following ordered indexing methods were investigated:

- We implemented the classical B-tree algorithm as in [4]. Then we experimentally tuned the B-tree node size to minimize cache misses, which on our hardware happened when each B-tree node contained 750 bytes.
- The CSB+ tree implementation was downloaded from [20]. We used the full CSB+ tree variant, which is the best variant for high insertion and update rates according to the authors.
- We implemented the burst trie index as specified in [10] adapted for integer keys.
- Judy was downloaded from [3].
- To have fair comparisons, in all indexing structures both keys and values are 32 bit integers.

The C code for indexing methods used in the experiments is available at [21].

We performed two sets of experiments with different key distributions. The first key distribution consists of uniformly distributed random integers from the whole 32 bit integer range. This is the worst case for tries since under this key distribution the trie structure will have sparse pointer arrays with many null pointers. In our second key distribution the keys are the position reports from the more realistic LRB input stream.

Given that the intention was to compare the scalability of indexing structures, the size of the indexes was gradually increased in a number of steps, and then the required time to perform insertion, single element retrieval, and range search operations were measured. After the each step the amount of main memory so far used by each index is noted. Moreover, to typify the measurement, the operations were done in batch, i.e. instead of measuring the time of inserting a single key in each step, which could be affected by noise, we calculated the average time to insert an element over 0.5 million keys. Since we measure pure insertion, update, and retrieval time duplicated keys in the input are omitted.

The performance of deletion is measured in a separate experiment.

4.1.1 Random key distribution

In this experiment first 16 million random keys from a flat distribution of integers in range $[0,2^{32}-1]$ were generated and stored in a one dimensional array. The experiments were performed by reading the keys from the flat array as follows:

In steps of size 0.5 million simulated incoming tuples, perform the following actions and measure the time each one takes on all indexing structures:

1. Insert into the index 0.5 million keys and measure the average time to insert one key.
2. Measure the accumulated amount of main memory used by each indexing structure after each 0.5 million inserts.
3. Retrieve in random order 50000 keys from all so far inserted keys and measure the average time to retrieve one key from the index.
4. Generate a random interval covering 10% of the total domain and make one range search to measure the time.

4.1.2 LRB key distribution

For a realistic data distribution, the indexing keys in the second experiment were LRB position reports. To construct ordered preserving integer keys *k* for LRB, they are computed as follows:

k= $VID+SEG*2^{20}+D*2^{27}+X*2^{29}$ i.e.:

- Expressway number *X*, the most significant 4 bits (28-31).
- Direction *D*, bit 27.
- Segment *SEG*, bits 20-26.
- Vehicle identifier *VID*, bits 0-19.

First the whole LRB input file is scanned; the first appearance of each key *k* is stored in a flat array. During this preprocessing, duplicates are detected and discarded.

After forming the flat array containing unique keys, the test is executed in a number of steps similar to the procedure for random keys, but steps three and four are different:

- In step three, query Q1 is executed for 50000 randomly chosen so far inserted position reports and the average single element retrieval time is measured.
- In step four, for randomly chosen *x, s,* and *d,* query Q2 is executed 100 times and the average search time measured.

4.1.3 Deletion

In order to compare the scalability of incremental window deletion with PTWI, we performed two tests on the two indexing structures Judy and B-trees.

The first test measures a naive incremental deletion strategy. In this test the indexes were loaded with different populations of keys as in 4.1.2. For each population all individual keys are deleted one by one, until the index becomes completely empty. The total time to empty an index with a given population is measured.

To measure deletion with PTWI, indexes with the same sizes as in the first test were populated, but this time, in contrast to deleting individual keys as in the first test, the whole index structure is dropped at once by traversing all nodes in it.

In both tests, to avoid memory fragmentation issues from biasing the performance, the application is restarted before any new index is created, i.e. before any new population is examined.

All experiments were run under Windows 7 on an Intel (R) Core(TM) i5 760 @2.80GHz 2.93 GHz CPU with 4GB RAM, single threaded using the Visual Studio 10 32 bits C compiler.

4.2 Experimental results

In this section the experimental results from comparing the indexing structures w.r.t. insertion, single element retrieval, memory utilization, range search, and deletion are analyzed. Experimental results of each indexing operation are presented and discussed under LRB key distribution alongside the random distribution. In the deletion section only LRB key distribution is presented since the results from random key distribution leads to the same conclusions.

4.2.1 Insertion

Figure 4 illustrates the time required to insert a single key into each indexing structure at a given index size when keys from LRB are used. The time is averaged over 0.5 million insertions.

The most important observation is that, in this key distribution, after around 4 million keys, Judy and burst tries reach their maximum depth of 4. Recall that in our experiments the keys are 4 byte integers and therefore their overall structure stabilizes. From this point on, it takes constant time to insert new elements into Judy and burst tries. The reason burst tries are faster than Judy is that their containers are not compressed, so insertion into them is simpler and computationally cheaper compared to Judy, where insertion into containers causes nodes to transform their representations.

As expected, B-trees and CSB+ trees scale logarithmic. CBS+ trees outperform B-trees w.r.t. insertion because of two reasons: First, all siblings of a node in a full CSB+ tree are allocated as soon the node is created, which reduces the costs of future node creation and potential structural balancing. Second, the CBS+ tree representation is cache conscious. However, after 8 million keys there is no more memory available in our 32 bit representation for the full CSB+ tree to grow.

Figure 5 illustrates the time required to insert a single key into each indexing structure at a given index size when keys are picked from a random distribution. The time is averaged over 0.5 million insertions.

As expected, B-trees and CSB+ trees scale logarithmic and show no sensitivity to the key distribution.

Under the random key distribution, in particular burst tries and, to a lower extent, Judy undergo an unstable period when the size is around 3 and 2 million keys, respectively. The fluctuation is due to the specific conditions under which bursting happens. That is, since the keys are uniformly distributed within a very wide range, they rarely share prefix at the next level, so during the burst, new containers are created for most of the keys that are being re-distributed. The high computation and memory management costs involved results in poor performance during bursting.

It is worth to note that Judy stabilizes much earlier than the burst trie. This is because Judy maintains dynamic container structures depending on the population of each sub-expanse, which decreases the bursting cost.

4.2.2 *Single element retrieval*

Figure 6 illustrates the time required to retrieve a single key from each indexing structure at a given index size under the LRB key distribution. B-trees and CSB+ trees scale logarithmic as expected, with CSB+ trees being faster mainly because of cache awareness. Retrieval of single elements from Judy and the burst trie takes constant time after they reach their maximum depth at four levels. Judy is fastest due to two reasons: first its efficient compression techniques facilitate search in the nodes. For example, bitmap nodes store only populated sub-expanses and index them using a directly accessed bitmap. Second, it exploits the CPU cache more efficiently.

Figure 7 illustrates the time required to retrieve a single key from each indexing structure at a given index size under the random key distribution. Again, since B-trees and CSB+ trees are not sensitive to key distribution, they scale similar to the LRB key distribution in Figure 6. The search time for burst tries increases until a maximum at around 3 million keys, after which the retrieval performance improves. The peak happens almost right before the nodes burst. Before the bursting happens, most of the containers are highly populated and therefore performing binary search in them becomes costly. After the bursting happens, keys are distributed among containers with lower populations and therefore the cost for binary search decreases. In Judy, in contrast to burst tries, due to the maintenance of dynamic population-based node structures, the search time at each node is optimized and therefore Judy is much more stable than burst tries.

4.2.3 *Memory utilization*

Figure 8 and 10 show the memory utilization with LRB and random distributions, respectively. The inefficient memory utilization of the CSB+ tree implementation is displayed separately in Figure 9.

To illustrate the compactness of indexing structures the main memory utilization is measured and displayed in terms of byte per key-value-pair (B/KVP). As a theoretical base line we also plot **flat arrays** in which KVPs are stored un-indexed in consecutive memory cells. Since all investigated indexing structures use 32 bit integers for both keys and value-pointers, storing KVPs in such a flat array –independent from the number of KVPs- achieves 8 B/KVP, the minimum uncompressed memory area needed to store key-value pairs.

Figure 8 shows the memory utilization when the LRB key distribution is used. It is worth to note that after a certain population, Judy becomes even more space efficient than flat arrays. The reason is that as the LRB simulation proceeds, the traffic increases. This means more vehicle ids per segments of expressways and consequently a more dense key distribution. Judy's dynamic node structure utilizes this to minimize memory consumption mainly through using bitmap nodes and leafs [18].

The burst trie on the other hand improves the memory utilization up to a point where the number of keys in sub-expanses exceeds the maximum container size and the containers burst. Each bursting brings about an additional node, and many new containers, which leads to bad memory utilization.

The memory utilization of B-trees is very stable but not as compact as Judy.

Figure 9 shows the memory utilization for CSB+ tree compared to the B-tree for the LRB distribution. The main reason for poor memory utilization of the Full CSB+ tree is that, as described in [12], it creates all sibling nodes for each node to improve insertion and update time. As the total number of keys increases, the memory utilization improves, but since creating new nodes is essential, and all the sibling nodes are also allocated at the time of creating any new node, the improvement is limited.

Figure 10 shows the memory utilization when random key distribution is used. As expected, since B-trees are insensitive to the key distribution, they make the same B/KVP as in Figure 8.

Under random key distribution, burst tries go through extreme bursting and they have even worse memory utilization. This is due to that most containers created by the burst for a random key distribution contain a single element. As the experiment proceeds, more keys with the same prefix are added to the newly created containers, and therefore memory utilization of burst tries improves.

Judy shows very little sensitivity to random key distributions, because in Judy containers with very low populations are implemented using a very specific compact structure - the immediate pointers. In immediate pointers the contents of nodes and leafs with one or two keys are stored in the pointer itself. The immediate pointers make the memory utilization become stable very soon. With random key distribution Judy's memory utilization is slightly worse than flat array and substantially better than the other indexing structures.

4.2.4 *Range search*

Figure 11 illustrates for LRB key distribution the time required to perform a range search at a given index size using B-trees, burst tries, CSB+ trees, the original iterator based Judy implementation, and Judy extended with the mapper for range search. As expected, our Judy mapper outperforms the original range search provided by Judy and it performs almost as efficient as a B-tree. The reason for the slightly better performance of the B-tree range search over Judy is that in B-trees a single node stores more keys compared to Judy for two reasons. First, Judy utilizes the most compact representation at each node which leads to nodes with lower populations. Second, compared to the rather large nodes in our B-tree, nodes in Judy are generally smaller since they need to fit in a 64 byte cache line.

Figure 12 illustrates for the random key distribution the time required to perform a range search using original and extended Judy, B-trees, CSB+ trees, and burst tries at a given index size. The range search scales worse using Judy compared to B-trees with random key distribution, because Judy creates huge number of very small *immediate* nodes.

In both Figure 11 and Figure 12, the Judy mapper implementation scales better than the burst trie mapper. The reason is that the internal nodes in burst tries include many null pointers, which increases the traversal time. That is, the internal nodes in burst tries always contain 256 pointers, even though many of them represent empty sub-expanses, and consequently, the burst trie mapper needs to consider all sub-expanses within the [low-high] range, including the empty ones. The more compact representation of Judy avoids many unnecessary memory accesses and cache misses. It should also be noted that the CSB+ trees in both cases are not faster than B-trees for range search. The main reason is the larger node size of B-trees, which utilizes the CPU cache better since the leaf nodes are very short in CSB+ trees compared to B-trees.

4.2.5 Deletion

Figure 13 measures PTWI performance using B-tree and Judy subwindow indexes. As expected, removing an index structure by deleting elements one by one, as in the incremental deletion, requires much more time than dropping of the whole indexing structure, as in PTWI. Notice that dropping a Judy index of a given size takes more time than dropping a B-tree index of the same size. This is due to the higher number of nodes in Judy, which needs more time to be traversed and freed. However, since in PTWI it is possible to perform the index drop operation at a background thread in a lazy manner, slight performance improvements in the index drop operation has insignificant contribution to the overall performance of a DSMS.

4.2.6 Discussion on experimental results

The following main aspects are involved in selecting the suitable ordered indexing structure for window based data streams: insert time, retrieval time, range search time, deletion time, and memory utilization.

Since in many DSMS applications, the input stream is massive, supporting scalable insertion is the most important aspect of indexing structures for sliding windows of data streams. From this perspective, a combination of constant and stable insert time is desired.

As important as insert scalability is scalability of deletions, which was addressed by PTWI. With the PTWI the scalability of deletion becomes non-critical.

The next important aspect is scalability of single element retrievals. To meet the real time requirements of continuous query processing, constant and stable retrieval time is essential.

Maintaining an index over a sliding window of massive streams requires indexing structures with acceptable memory utilization. An indexing structure with poor memory utilization restricts the stream rate that can be handled by the DSMS.

Finally in many applications, e.g. computer network and urban traffic monitoring applications, range search is needed for answering ad hoc queries from the current window.

Table 1 summarizes the results of our experiments in a qualitative manner. For memory utilization Judy is clearly the best because of its extensive compression. Judy is also the best on insertion and single key accesses. For range searches the tree based indexing methods are the best, but the extended Judy is very close in particular for non-random data. Burst tries are not stable in general and therefore problematic for streaming applications. The main disadvantage with Judy is its very complex implementation.

To conclude, in the context of indexing sliding windows of streams, our extended version of Judy outperforms all other indexing structures.

Table 1. Qualitative summary of the experimental results.

	B- tree	CSB+ tree	Burst trie	Judy	Extended Judy
memory utilization	good	worst	bad	best	best
insertion	good	good	best	best	best
key access	good	good	good	best	best
range search	best	best	bad	worst	good
stability	best	good	worst	best	best
simplicity	best	bad	best	worst	worst

5. Conclusions and future work

Window based ordered indexing of data streams has additional requirements to traditional DBMS indexing. Other than scalable range search and individual element retrieval, it is essential for DSMS indexing structures to support scalable insertion and deletion under high volume stream rates. We investigated a number of data stream indexing methods by implementing them or extending available implementations. We compared the following index structures: B-trees, burst tries, CSB+ trees, and the advanced Judy implementation of compact tries.

Through extensive experiments we showed that, in the context of window based indexing of data streams, compact tries with improved range search capabilities outperform other investigated methods by consuming much less main memory, supporting constant access time for insertion and retrieval, and being capable of performing scalable range search. The combination of the above characteristics makes extended compact tries the best ordered indexing method for window based stream processing.

The performance of massive deletion is also very important when indexing high volume data stream windows, which was addressed by PTWI.

Highly tuned scalable indexing structures like Judy might become extremely complex, making any modification very costly. Therefore, we devised non-intrusive methods to add functionality to an existing index structure. PTWI and mappers are two examples of nonintrusive additions that enabled us to successfully re-use, improve, and integrate industry strength software.

In particular, our extended version of Judy with mapper-based range search and PTWI outperforms the other investigated indexing structures, making it attractive for sliding stream window indexing in general.

In addition to supporting range search, tries support more general pattern matching operations that might be useful in pattern recognition in data streams. Therefore, adding pattern matching to tries will be a valuable future work.

6. ACKNOWLEDGEMENTS

This work was supported by the Swedish Foundation for Strategic Research, grant RIT08-0041 and by the EU FP7 project Smart Vortex.

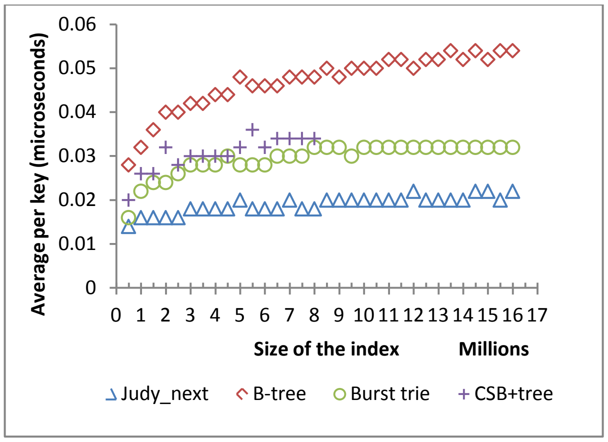

Figure 6. Single element retrieval under LRB key distribution

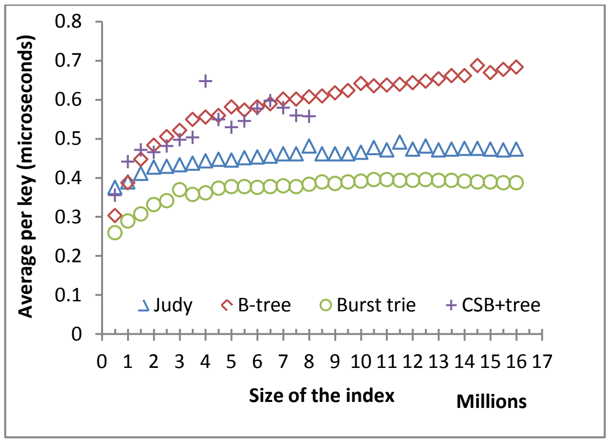

Figure 4. Insertion under LRB key distribution

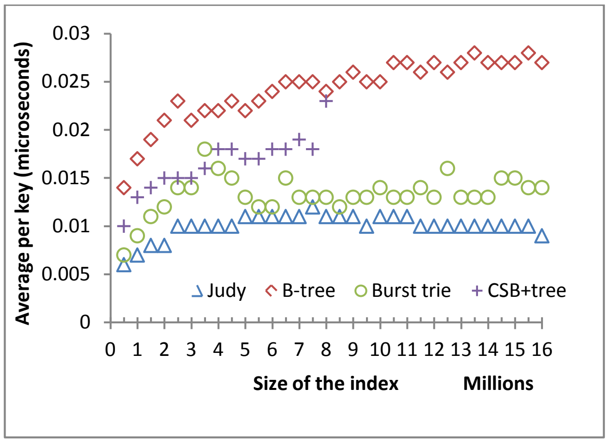

Figure 7. Single element retrieval under random key distribution

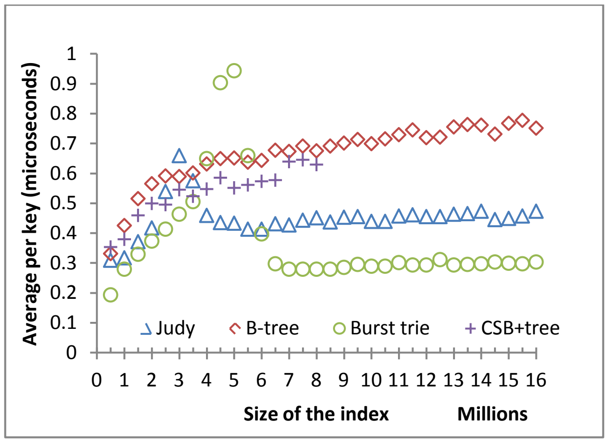

Figure 5. Insertion under random key distribution

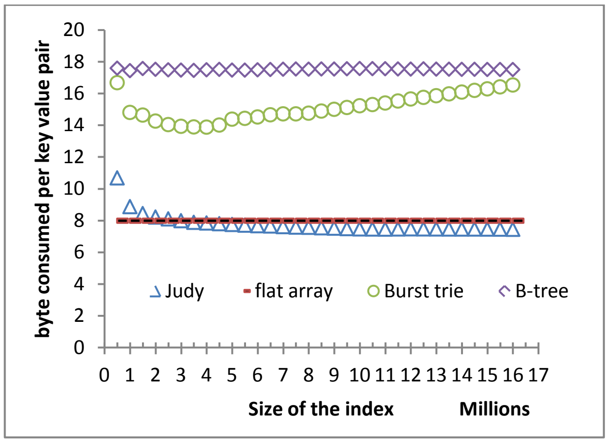

Figure 8. Memory utilization under LRB key distribution

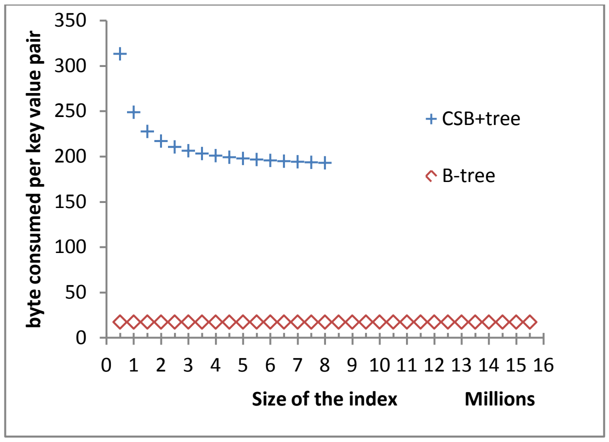

Figure 9. Memory utilization of full CSB+trees

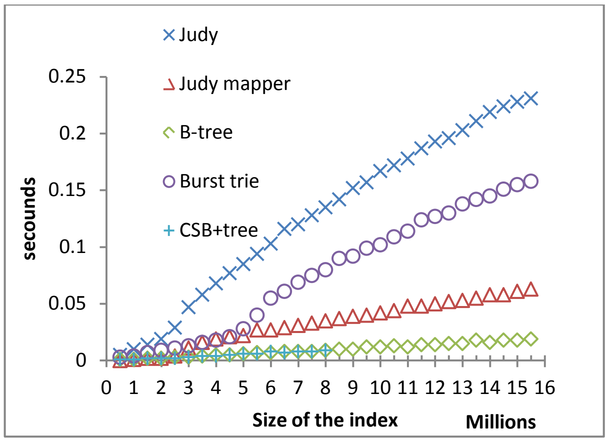

Figure 12. Range search under random key distribution

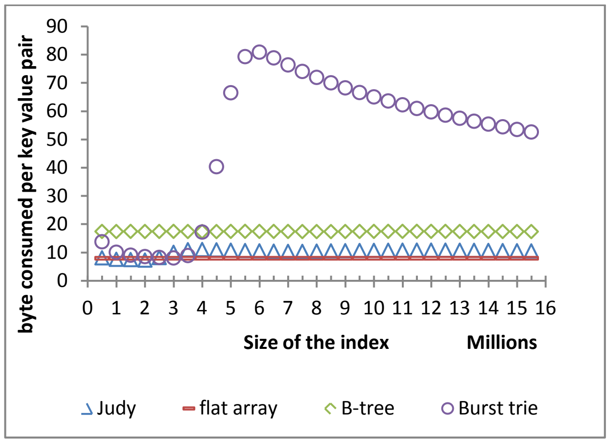

Figure 10. Memory utilization under random key distribution

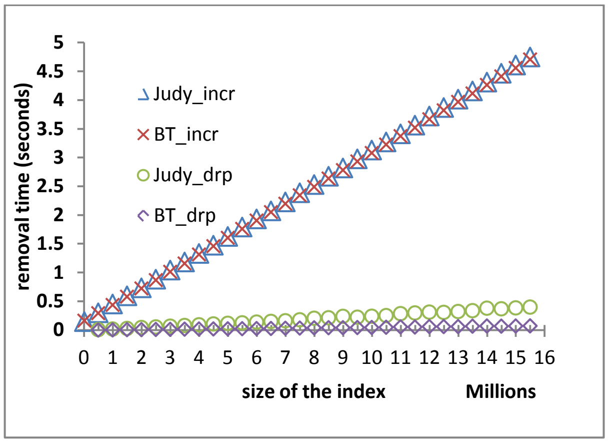

Figure 13. Incremental deletion vs. PTWI

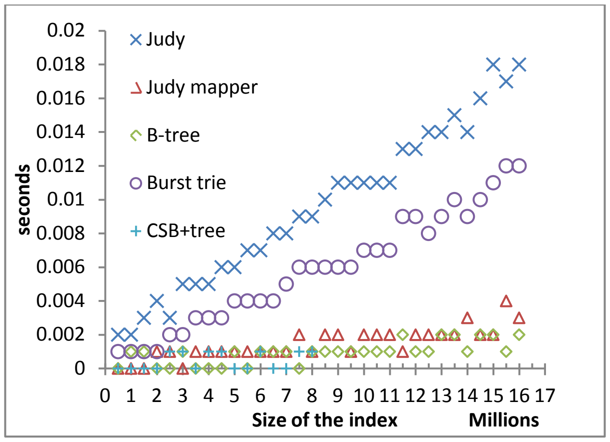

Figure 11. Range search under LRB key distribution

7. REFERENCES

1 Arasu, A., Cherniack, M., Galvez, E. et al. Linear road: a stream data management benchmark. In *VLDB '04 Proceedings of the Thirtieth international conference on Very large data bases* (2004).

2 Babu, S. and Widom, J. Continuous queries over data streams. *ACM SIGMOD Record*, 30, 3 (2001), 109-120.

3 Baskins, D. Judy home page [http://judy.sourceforge.net/] (2003).

4 Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1 (1972), 173–189.

5 Bentley, J. L. and Sedgewick, R. Fast algorithms for sorting and searching strings. In *SODA '97 Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms* (1997).

6 Boehm, M., Schlegel, B., Volk, P. B., Fischer, U., Habich, D., and Lehner, W. *Efficient In-Memory Indexing with Generalized Prefix Trees*. 2011.

7 Graefe, G. B-tree indexes for high update rates. *ACM SIGMOD Record*, 35, 1 (2006), 39 - 44.

8 Graefe, G. Volcano-an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6, 1 (1994), 120-135.

9 Hankins, R. A. and Patel, J. M. Effect of node size on the performance of cache-conscious B+-trees. In *Proceedings of the 2003 ACM SIGMETRICS* (2003).

10 Heinz, S., Zobel, J., and Williams, H. E. Burst tries: a fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20, 2 (2002), 192 - 223.

11 Hellerstein, J. M., Naughton, J. F., and Pfeffer, A. Generalized Search Trees for Database Systems. In *Proceedings of the 21st VLDB Conference* (Zurich, Switzerland 1995).

12 Jun R., Kenneth A. R. Making B+- trees cache conscious in main memory. In *MOD* (Dallas TX 2000), Proceedings of the 2000 ACM SIGMOD international conference on Management of data.

13 Kurtz, S. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, 29 (1999), 1149--1171.

14 Lehman, T. J. and Carey, M. J. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the 12th VLDB Conference* (1986), Proceedings of the Twelfth International Conference on Very Large Data Bases.

15 Luan, H., Du, X., and Wang, S. Prefetching J+-Tree: A Cache-Optimized Main Memory Database Index Structure. *Journal of Computer Science and Technology*, 24, 4 (2009), 687-707.

16 Luan, H., Du, X., Wang, S., Ni, Y., and Chen, Q. J+ -Tree: A New Index Structure in Main Memory. In *Advances in Databases: Concepts, Systems and Applications*. Springer Berlin, Heidelberg, 2007.

17 Oracle®. Oracle® Database VLDB and Partitioning Guide. *Available at http://docs.oracle.com/cd/B28359_01/server.111/b32024/part_admin.htm*.

18 Silverstein, A. Judy IV Shop Manual. *Available at http://judy.sourceforge.net/doc/shop_interm.pdf* (2002).

19 Tatbul, N. and Zdonik, S. Window-aware load shedding for aggregation queries over data streams. In *VLDB '06 Proceedings of the 32nd international conference on Very large data bases* (2006).

20 http://www.cs.columbia.edu/~kar/software/csb+/

21 http:/www.it.uu.se/research/group/udbl/DSMSOrderedIndexing