

# Indexing and Querying Constantly Evolving Data Using Time Series Analysis <sup>\*</sup>

Yuni Xia<sup>1</sup>, Sunil Prabhakar<sup>1</sup>, Jianzhong Sun<sup>2</sup>, and Shan Lei<sup>1</sup>

<sup>1</sup> Computer Science Department, Purdue University

<sup>2</sup> Mathematics Department, Purdue University

{xia, sunil, leishan}@cs.purdue.edu, sunj@purdue.edu

**Abstract.** This paper introduces a new approach for efficiently indexing and querying constantly evolving data. Traditional data index structures suffer from frequent updating cost and result in unsatisfactory performance when data changes constantly. Existing approaches try to reduce index updating cost by using a simple linear or recursive function to define the data evolution, however, in many applications, the data evolution is far too complex to be accurately described by a simple function. We propose to take each constantly evolving data as a time series and use the ARIMA (Autoregressive Integrated Moving Average) methodology to analyze and model it. The model enables making effective forecasts for the data. The index is developed based on the forecasting intervals. As long as the data changes within its corresponding forecasting interval, only its current value in the leaf node needs to be updated and no further update needs to be done to the index structure. The model parameters and the index structure can be dynamically adjusted. Experiments show that the forecasting interval index (FI-Index) significantly outperforms traditional indexes in a high updating environment.

## 1 Introduction

Constantly evolving data arises in numerous applications, for example, a moving objects database stores the current positions for millions of moving objects and these data change frequently over the time. A stock database stores the latest quotes for a large collection of stocks and they vary by minute or even second. The constant changing nature of the data brings challenges to a wide range of issues such as data storing, indexing, querying, mining and so on. In this paper, we focus on indexing and querying constantly evolving data.

Existing dynamic index structures perform satisfactorily for traditional database applications where updates are infrequent in comparison to queries. They are designed mainly for the purpose of efficiently supporting query processing. For evolving data applications that are characterized by numerous and frequent data updates, these indexes suffer from high updating overhead and result in poor performance. In order to reduce the index updating cost, most existing approaches

---

<sup>\*</sup> Portions of this work were supported by NSF CAREER grant IIS-9985019, NSF grant 0010044-CCR, NSF grant 9988339-CCR

use a simple linear or recursive function to describe the data changing patterns. However, the changing patterns in many situations (for example the stock prices) are too complex to be described by a simple function that changes infrequently.

In this paper, we propose to use time series analysis techniques to model and forecast constantly evolving data. We choose Box-Jenkins's autoregressive integrated moving average (ARIMA) methodology to identify the models, estimate model parameters and make N-step ahead predictions for each data. The index is built based on the forecasting interval for each data. Since the ARIMA model can efficiently capture the patterns of the data evolutions, the forecasting intervals are expected to be accurate and tight. An index built based on the forecasting intervals can accommodate data evolutions and substantially reduce index updating cost.

The rest of the papers proceed as following: section 2 discusses the related work for constantly evolving data indexing. In section 3, we propose a framework to use ARIMA technique to model and forecast the data and develop index based on the predicated intervals. We also explain the index details including its construction, updating and query processing. In section 4, we propose a mathematical model to determine the optimal interval size by balancing the indexing updating and querying cost. Experimental evaluation of the proposed approach is presented in section 5 and section 6 concludes the paper.

## 2 Related Work

Developing efficient index structures for constantly evolving data is an important research issue of databases. Most works in this area so far focus on moving object environment, where the positions of objects keep changing. As a simple approach, multi-dimensional spatial index structures can be used for indexing the positions of moving objects, however, they are not efficient because of frequent and numerous update operations. To reduce the number of updates, many approaches describe the moving object location by a linear function. Saltenis et al. [1] proposed the time-parameterized R-tree (TPR-tree). In this scheme, the position of a moving point was represented by a reference position and a corresponding velocity vector. Later, Tao et al [2] presented TPR\*, which extends the idea of TPR-trees by employing a different set of insertion and deletion algorithms in order to minimize the query cost. Recently, Tao et al [3] proposed a novel recursive motion function to support a broad class of non-linear motion patterns. They also proposed a general client-server architecture for answering typical spatio-temporal queries and STP-tree for indexing the expected trajectories. Kollios et al. [4] proposed an efficient indexing scheme using partition trees. Tayeb et al. [5] introduced the issue of indexing moving objects to query the present and future positions and proposed PMR-Quadtree for indexing moving objects. Agarwal et al.[6] proposed various schemes based on the duality and developed an efficient indexing scheme to answer approximate nearest-neighbor queries. All these techniques use linear or recursive functions to describe the

data changing patterns, however, in many applications, the data evolutions are far more complicated to be defined by simple linear or recursive functions.

Other techniques have been proposed to reduce the index updating cost. In [7], Kwon propose the lazy Rtree. The index structure is updated in a lazy way that is, if the point is still within the current MBR, then just update the old position to new position and no further update is needed to be done to the index structure. Only when the new position is out of the current MBR, the old position should be deleted and the new position should be inserted into the index. In [8], a bottom-up approach is proposed to improve the updating performance. The strategy improves the robustness of R-trees by supporting different levels of index reorganization ranging from local to global during updates, thus using expensive top-down updates only when necessary.

In time series area, numerous work had been done on various issues including classification, clustering, representation, anomaly detection, similarity-based query, whole-sequence and sub-sequence matching, time sequence indexing, statistical monitoring and so on. Our work distinguishes itself from above in that we go beyond the idea of time series modeling and forecasting. We use the modeling and forecasting intervals to develop an index for the constantly evolving data and effectively reduce index updating cost and improve the index performance.

### 3 Indexing Constantly Evolving Data

In this section, we explain the time series modeling and forecasting process and the details of the index construction, updating and querying procedures.

Our index uses the lazy-update Rtree [7], as shown in figure 1. It consists of two components, a regular Rtree as the primary index and an secondary index mapping the data ID to its page in the primary index. With the secondary index, it takes constant I/O to find the page for each data given its ID. In practice, the secondary index is usually put in memory. When the new value  $V_{i_{new}}$  for data  $i$  arrives, The index is updated in a lazy way that if  $V_{i_{new}}$  is still within the MBR of the old value  $V_{i_{old}}$ , then just update  $V_{i_{old}}$  to  $V_{i_{new}}$  and no further update is need to be done to the index structure. Only when  $V_{i_{new}}$  is out of the MBR of  $V_{i_{old}}$ ,  $V_{i_{old}}$  is deleted and  $V_{i_{new}}$  is inserted to the index.

We propose to take each evolving data as a time series and use time series analysis tools to identify the model for each data based on its history and make  $n$ -step ahead forecasts  $(V_1, V_2, \dots, V_n)$  for it. The forecasting interval (FI) is the smallest interval  $(I_{low}, I_{high})$  that contains  $(V_1, V_2, \dots, V_n)$ . An Rtree index is built based on the forecasting intervals for each data. We call it the Forecasting Interval Rtree (FI-Rtree). The leaf node of the FI-Rtree stores both the forecasting interval and the current value for each data. When the new value  $V_{i_{new}}$  for data  $i$  arrives, it is checked against its forecasting interval  $(I_{low}, I_{high})$ , if it is within  $(I_{low}, I_{high})$ , just update its old value  $V_{i_{old}}$  in the leaf node to  $V_{i_{new}}$  and no further update need to be done to the index structure. Only when  $V_{i_{new}}$  is out of the interval  $(I_{low}, I_{high})$ , the interval needs to be adjusted and the index should be updated.

The time series modeling tool is time or error triggered, which means it is triggered to run at certain time interval, for example, every n minutes, or when the newly arriving data values are different from the forecasted intervals by some thresholds. When the newly arriving data are quite different from the forecasted interval, it could be an indication that the data evolving patterns changes, therefore, the time series modeling process should be rerun by taking the recently history into consideration and determine if the model should be changed or the model parameters should be adjusted. The framework we propose for indexing constantly evolving data is shown in figure 2.

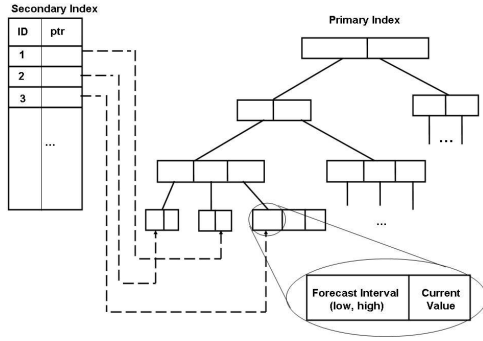


Fig. 1. Indexing Constantly Evolving Data

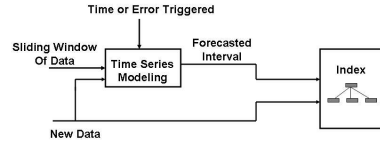


Fig. 2. FI-Rtree Index Structure

### 3.1 Time Series Modeling

A discrete time series is a set of time-ordered data  $(x_{t_1}, x_{t_2}, \dots, x_{t_n})$  obtained from observations of some phenomenon over time. An intrinsic feature of a time series is that typically, adjacent observations are dependent. We choose ARIMA for time series modeling because it covers a wide variety of patterns, including: *stationary* time series, which is in statistical equilibrium and fluctuates around a constant mean with constant variance. *non-stationary* time series, which has no natural mean, but tends to increase or decrease over time. *seasonal* time series, which repeat at regular intervals. Stationary series are described by Autoregressive Moving Average ARMA(p,q) models, non-stationary series by Autoregressive Integrated Moving Average ARIMA(p,d,q) models and seasonal series by  $ARIMA(p, d, q) \times (P, D, Q)^S$  multiplicative models.

An ARMA(p,q) model captures two types of correlation via two main components, show in the following Equation:

$$X_t = \phi_0 + \phi_1 x_{(t-1)} + \phi_2 x_{(t-2)} + \dots + \phi_p x_{(t-p)} + \epsilon_t + \theta_1 \epsilon_{(t-1)} + \theta_2 \epsilon_{(t-2)} + \dots + \theta_q \epsilon_{(t-q)}$$

The autoregressive (AR) component describes the most p significant correlations between the current observation  $X(t)$  and the past observations  $x(t-1)$ , ...,

$x(t-p)$ . Here, we assume observations that are close together are more likely to be correlated than those are far apart. The moving average (MA) component describes the most  $q$  significant correlations between the current observations  $y(t)$  and the past noise terms  $e(t-1), \dots, e(t-q)$ . These noise represents uncertainty and are used to estimate the non-deterministic characteristics in time series. They are assumed to be normally distributed with zero mean and constant variance.  $(\phi_1, \phi_2, \dots, \phi_p)$  and  $(\theta_1, \theta_2, \dots, \theta_q)$  are autoregressive (AR) coefficients and moving average (MA) coefficients respectively and they express the magnitudes of correlations.

The arma(p,q) model can be extended to be the arima(p,d,q) model by inserting the number of differencing transformations  $d$ . It indicates that, after transforming the series  $d$  times, the final series will be stationary and will have  $p$  autoregressive terms and  $q$  moving average terms. Seasonal series can also be transformed via differencing to become stationary, just similar to non-stationary series. Here, the differencing interval (i.e. the gap between two differenced observations) is the season length  $S$ . To represent seasonal series, the arima(p, d, q) model can be further extended to include a seasonal component  $(P, D, Q)^S$ . This component specifies that after  $D$  seasonal differencing with a season length  $S$ , the correlation structure among those differenced observation pairs that are separated by  $S$  is stationary, with  $P$  autoregressive and  $Q$  moving average terms. The  $ARIMA(p, d, q) \times (P, D, Q)^S$  model is also known as the general ARIMA model because it can represent all three types of series, namely stationary, non-stationary and seasonal and their combinations. Note that ARIMA(p, 0, q) X(0, 0, 0) model is simply the ARMA(p, q).

The usual approach to fit an ARIMA model to a time series includes three steps:

1. Model Identification: There are two important parameters for identifying time series, Autocorrelation Function (ACF)  $\rho_k$  and Partial Autocorrelations (PACF)  $\phi_{kk}$ . ACF is represented as a plot of the autocorrelation as a function of lag. The autocorrelation is simply the correlation of a time series with itself at a specified lag. The partial autocorrelation (PACF) at a given lag is the autocorrelation that is not accounted for by autocorrelations at shorter lags. Details of computing ACF and PACF can be found in [9].

2. Parameter Estimation: Once a model is identified, the next step is to estimate the parameters, which are the magnitudes of the  $p$  and  $P$  autoregressive terms and the  $q$  and  $Q$  moving average terms. To find the correlation magnitude, also known as the model coefficients, common technique include maximum likelihood estimation and least squares. We choose the least square approach.

3. Forecasting: Based on the estimated parameters and the model structure, we can produce  $n$ -step ahead forecasts  $(\hat{y}(t+1), \hat{y}(t+2), \dots, \hat{y}(t+n))$  for each data. The  $n$ -step ahead forecasting interval for each data is  $(I_{low}, I_{high})$ , in which  $I_{low} = \min(\hat{y}(t+1), \hat{y}(t+2), \dots, \hat{y}(t+n))$  and  $I_{high} = \max(\hat{y}(t+1), \hat{y}(t+2), \dots, \hat{y}(t+n))$ .

### 3.2 Index Construction

Assume there are  $m$  constantly evolving data  $(o_1, o_2, \dots, o_m)$  in the database, after the timer series modeling and forecasting, we obtain the  $n$ -step ahead predictions intervals for each data  $(I_1, I_2, \dots, I_m)$ . The index is then built based on these intervals. The leaf nodes of the index contain both the actual values as well as the forecasting intervals for each data and the MBRs of the index are computed based on the forecasting intervals instead of the actual data values. Please note that when history data is not available, we can still make simple initial forecast for each data. Although the initial forecasting interval might be inaccurate in the beginning, as data stream in and more history data become available for modeling, the forecasting intervals will become more accurate.

As mentioned earlier, the index we propose is a lazy-update Rtree with a secondary index. Each entry in the secondary index maps a data ID to its page number in the Rtree. Therefore, for each data ID, we put in the secondary index the corresponding page number which identifies the page in the Rtree that contains that data.

### 3.3 Index Update

When a data changes and the new value arrives, first, the secondary index is looked up to find out its the page number in the Rtree corresponding to that data. According to the page number, we can immediately retrieve the page that contains that data and find out its old value and its forecasting interval. If the new data value is still within its forecasting interval, the old value is updated to the new value and no further update is needed to be done to the index structure. This is because that the MBRs of the FI-index is computed based on the forecasting intervals instead of the accurate data values. Thus as long as the data changes within its forecasting interval, the index structure remains correct and no need for updating.

When the new data value moves out of its forecasting interval, we will check if it deviates from the forecasting interval by a certain threshold, or if the ForecastMissCount has reached a certain threshold (ForecastMissCount is a counter that records how many times the forecasting interval fail to accommodate the data changes). All these indicate that the time series model for that data may be outdated. In that case, the time series modeling and forecasting process is triggered to rerun by taking the recent history into consideration and the new forecasting interval is obtained. If the new data value is out of the forecasting interval, but it does not deviate from it by a certain threshold; we will simply enlarge the forecasting interval to accommodate the new data value and increase the ForecastMissCount by 1, which means forecasting interval fails to enclose the data changes for one more time. The reason we keep an ForecastMissCount is that we do not hope one or two outliers or noise data trigger the time series modeling process to rerun. After the new forecasting interval is obtained, the Rtree index is updated in a lazy way, that is, if the new forecasting interval is within the MBR of the old forecasting interval, just update the old interval with the

new one and no further update to the index is needed. When the new forecasting interval is out of the MBR of the old interval, the old interval together with the old data value should be deleted, and the new forecasting interval with the new data value will be inserted. The pseudo code of the index updating procedure is given in the algorithm 1. Our index is developed to for the purpose of reducing index updating cost when data changes. Since we analyze and model the data evolutions and make effective forecasts and build index based on the forecasting intervals, the data are very likely to change within the forecasting intervals and do not incur additional update to the index structure.

---

**Algorithm 1** Index Updating Algorithm
 

---

```

1: Read in the new value  $V_{i-new}$  for data  $i$ 
2: Look up the secondary index and find the page number  $P_i$  for data  $i$ 
3: Read in Page  $P_i$  from the Primary Rtree index, find its interval
   ( $I_{old-low}, I_{old-high}, V_{i-old}$ )
4: if  $V_{i-new} \in (I_{old-low}, I_{old-high})$  then
5:   Update  $V_{i-old}$  to  $V_{i-new}$ 
6:   Return;
7: else
8:   if ( $V_{i-new} \leq I_{old-low} - Threshold$ ) or ( $V_{i-new} \geq I_{old-high} + Threshold$ ) or
   ForecastMissCount[i]  $\geq T$  then
9:     ( $I_{new-low}, I_{new-high}$ ) = TimeSeriesModeling()
10:    ForecastMissCount[i] = 0;
11:   else
12:     ( $I_{new-low}, I_{new-high}$ ) = ExpandInterval( $V_{i-new}$ )
13:    ForecastMissCount[i] ++;
14:   end if
15:   if ( $I_{new-low}, I_{new-high}$ ) is within the MBR of ( $I_{old-low}, I_{old-high}$ ) then
16:     Update ( $I_{old-low}, I_{old-high}, V_{i-old}$ ) with ( $I_{new-low}, I_{new-high}, V_{i-new}$ )
17:     Return;
18:   else
19:     Delete ( $I_{old-low}, I_{old-high}, V_{i-old}$ )
20:     Insert ( $I_{new-low}, I_{new-high}, V_{i-new}$ )
21:   end if
22: end if

```

---

### 3.4 Query Processing

In this section, we explain how the forecasting interval index (FI-Index) supports querying. We start with the range query. A range query (a, b) searches for all data items that falls within the interval (a,b). To process a range query, we first check the query (a,b) against MBRs of the nodes in this tree. A node N is pruned when it is guaranteed that no item in the subtree rooted at N can satisfy (a, b). Let  $I_{low-min}$  be the minimal of the interval lower ends over all the forecasting interval in the subtree of N and  $I_{high-max}$  be the maximal of all the interval

high ends in the subtree. Note that the MBR for N is  $[(I_{low-min}, I_{high-max})]$ . if the query (a, b) does not overlap with  $[(I_{low-min}, I_{high-max})]$ , we can say the no data items in the subtree of N overlap with query (a,b) and the subtree of N can be safely pruned. The reason is that the forecasting interval for each data covers the its current value, therefore, if the interval does not intersect with the query, then the data can not fall in the query range (a,b) either.

The same pruning approach can be used for other queries such as point queries and nearest neighbor queries. For nodes that can not be pruned, if it is a leaf node, all the data items it contained will be compared with the query; if it is an internal node, its children nodes should be retrieved and this pruning process will run recursively.

## 4 Optimal Interval size

In this section, we give an analysis to the performance of the FI-Index and give insight on how to determine the optimal interval size the index should choose. The disc I/O cost consists of two main components:

- Update: Both the current value and the old value of the data could be retrieved through the secondary index, based on the range stored in the corresponding page, we decide whether to update the tree or not. Hence the disc I/O cost is  $1 + \{\text{updating cost}\}$  if we need to update the tree, otherwise just 1, where the update tree cost includes the cost to search the object in the R-tree and the cost to insert the object into the R-tree. We use the following heuristic formula to calculate the average cost of the disc I/O:

$$1 + R(x)(\{\text{average updating cost}\}),$$

where  $R(x)$  is the probability that the update operation is a non-lazy one, that is, the probability that the new data value falls out of the interval. Obviously, this probability depends on the interval size  $x$ . The larger the interval size  $x$ , the smaller the probability. We assume  $R(x)$  is inversely proportional to  $x$ ,  $R(x) = \frac{\mu}{x+\lambda}$ . To make it cover the case of the traditional R-tree and the lazy updating cases, let  $R(0)$  correspond to the traditional R-tree which is built based on the current value of the data with interval size 0. The disc I/O cost for one update tree operation is usually proportional to the height of the tree, hence

$$\{\text{average updating cost}\} = (C_s + C_i) \lceil \log_F N_{obj} \rceil,$$

where  $F$  is the average fan-out,  $C_s$  is a constant depending on the search performance of the R-tree, and  $C_i$  is a constant depending on the insertion performance of the R-tree,  $C_s$  and  $C_i$  can be decided by experiments.

- Query: The cost of one range query is related to the size of the query and the search of the boundary data. Assume the total number of pages covered

in the average range is  $T$ , We use the following heuristic formula to bound the average cost of the range query disc I/O cost:

$$C_s Q(x) \left( T + \frac{T}{F} + \frac{T}{F^2} + \dots + 1 \right) = \frac{C_s Q(x) T}{1 - \frac{1}{F}}$$

where  $C_s$  is the search cost.  $T, \frac{T}{F}, \frac{T}{F^2}, \dots$  are the number of nodes need to be accessed in different levels.  $T$  is the number for the leaf level,  $\frac{T}{F}$  for the level above the leaf, and so on, till the root level, which is 1.  $Q(x)$  is an increasing function depends on the average variance of the total objects. Let  $Q(0) = 1$  correspond to the traditional R-tree. Obviously, the larger the intervals are, the less precise the index is, and the less efficient the index is for supporting query processing. We assume  $Q(x)$  is proportional to  $x$  and since  $Q(0) = 1$ , we define  $Q(x) = \kappa x + 1$ , where  $\kappa$  is a constant depending on the selectivity. { total pages covered in the average range } depends on the size of the query and the probability distribution of the objects.

To simplify the analysis, we assume the objects stay in  $[0, 1]$  follow the uniform distribution, and the moving of every object follows the normal distribution  $N(0, \bar{\sigma})$ , The average query size is assumed to be  $f_q$  in sense of the total length of the query interval, hence

$$\{ \text{total pages covered in the average range} \} = \lceil \frac{N_{obj} f_q}{F} \rceil.$$

The sum of the updating cost and query cost in term of disc I/O is

$$I_{I/O} = N_u (1 + R(x, u) (C_s + C_i)) \lceil \log_F N_{obj} \rceil + N_q C_s Q(x) \left( \frac{N_{obj} f_q}{F} + \frac{N_{obj} f_q}{F^2} + \dots + 1 \right)$$

Since  $R(x) = \frac{\mu}{x + \lambda}$  and  $Q(x) = \kappa x + 1$ , then

$$I_{I/O} = N_u \left( 1 + \frac{\mu}{x + \lambda} (C_s + C_i) \lceil \log_F N_{obj} \rceil + N_q C_s (\kappa x + 1) \left( \frac{N_{obj} f_q}{F} + \frac{N_{obj} f_q}{F^2} + \dots + 1 \right) \right)$$

To get the minimum value of  $I_{I/O}$ , we make  $\frac{dI}{dx} = 0$ , therefore

$$-N_u (C_s + C_i) \lceil \log_F N_{obj} \rceil \mu (x + \lambda)^{-2} + N_q C_s \kappa \left( \frac{N_{obj} f_q}{F} + \frac{N_{obj} f_q}{F^2} + \dots + 1 \right) = 0$$

. We obtain the optimal interval size for the data is:

$$x = \sqrt{\frac{N_u (C_s + C_i) \lceil \log_F N_{obj} \rceil \mu}{N_q C_s \kappa \left( \frac{N_{obj} f_q}{F} + \frac{N_{obj} f_q}{F^2} + \dots + 1 \right)}}$$

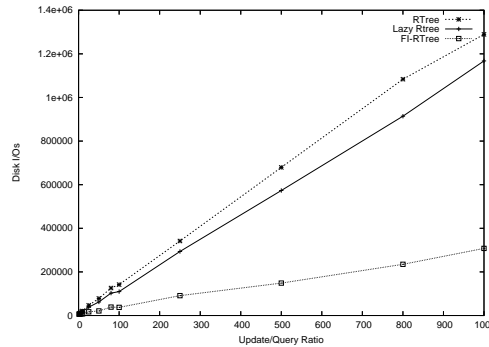
From the above formula, we can see that the optimal interval size is proportional to the number of updates and inversely proportional to the number of the queries. This is natural since the larger the intervals are, the less likely the data values will move out of the intervals and the less update needs to be done on the

index structure, however, the query performance will be worse due to the impreciseness of the data, which leads to less pruning. Therefore, when the number of the update operations is much larger than the query, the index should take large intervals, while when queries operations are dominant, the index should take smaller intervals.

## 5 Experimental Evaluation

Three index structures are evaluated in our experiments: the traditional R-tree; the lazy Rtree (The traditional R-tree augmented with lazy updating using the secondary index structure) and the FI-Rtree. Our experiments are based real stock data. 25,000 stocks time series, each of which contains over 200 data points are used in the experiments. The first 100 data points of each stock are used for ARIMA modeling and forecast. Based on the forecasting intervals, we build the FI-Rtree. Once the FI-Rtree is built, the remaining data points are modeled as dynamic updates to the FI-Rtree, as well as other R-tree variants. At the same time, a number of range queries are generated and evaluated. Each range query has its central location chosen randomly and has a query size a fraction of the price range.

Since these are disk-based index structures, the number of page I/Os is the natural metric for measuring the performance of the indexes. We measure the number of page I/Os for reads and writes of both dynamic updates and queries during the simulation. The secondary index of the LazyRtree and the FI-Rtree is assumed to be in the main memory. For the time series modeling, we used the TsModeler, an automatic ARIMA Time Series Modeling Tool. [10].



**Fig. 3.** Overall Disk I/Os

We study the relative performance of the various index structures as the relative number of queries and updates is varied. Figure 3 shows the total number of page I/Os performed for querying and updating the R-tree, the *lazy*-R-tree, and the FI-Rtree. The performance is measured under the same query generation

rate but different update arrival rates. As the ratio of update rate over the query rate is increased from 0 to 1000, all four indexes show an increase in the number of I/Os. This is because increasing the update rate implies more demands on the index, and consequently more I/Os are needed.

When the update/query ratio is very low, the FI-Rtree takes more I/Os than the other R-tree variants. The reason is that the R-tree and the *lazy*-R-tree uses actual data values, while the FI-Rtree employs data intervals and result in a worse query performance. Towards the right end of the graph, when the update workload dominates the query workload, the FI-Rtree has a significant improvement over other R-tree variants. In fact, the number of I/Os needed by all three R-trees increases sharply, whereas the FI-Rtree gracefully handles the high update burden. When updates are much more frequent than queries, which is a typical scenario in sensor and moving object databases, the R-tree suffers from expensive updates. The distinction between the R-tree and the *lazy*-R-tree begins to show in this high update setting as the secondary index yields significant gains from cheaper updates. The FI-Rtree clearly outperforms the other indexes in this high update environment since its structure is inherently designed to maximize tolerance to changes in data values. The advantage of better update performance more than compensates for the slightly poorer query performance. As the update/query ratio increases, the improvement of the FI-Rtree over R-trees is more obvious. In particular, when the update/query ratio is 1000, the number of I/Os required by the FI-Rtree is only one-fourth that of the *lazy* Rtree, and one-fifth that of the R-tree.

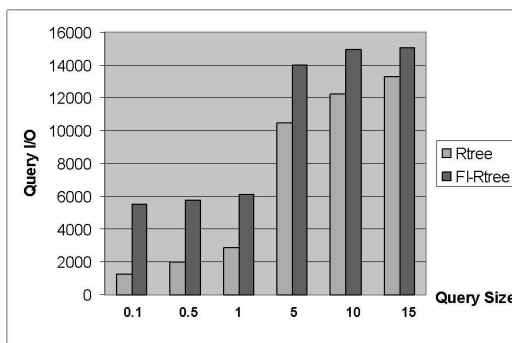


Fig. 4. Query I/Os vs. Query Size

We also studied how the query size affects the query performance of the FI-Rtree and the traditional Rtree. Note that since the *lazy*-R-tree and the traditional R-tree have almost identical query performance, here we compare the query I/Os of FI-Rtree with only the traditional Rtree. Figure 4 shows the query I/Os for FI-Rtree and traditional Rtree over different query sizes. The query size is varied from 0.1% to 15% of the domain. We find that the FI-Rtree always re-

quires more query I/Os than the traditional R-tree. However, as the query size increases, the performance of the FI-Rtree gets closer to that of the R-tree. The reason is that with a large query area, the probability that a given region will be covered by a query increases. Thus the advantage of having a small MBR is diminished with larger queries.

## 6 Conclusions and Future Work

This paper proposes a new approach for efficiently indexing and querying constantly evolving data. Each data is considered as a time series and we apply the ARIMA methodology to model it. The model enables making effective forecasts for the data and the index structure is developed based on the forecasting intervals. The model parameters and the index can be dynamically adjusted. Our experiments show that the FI-index significantly outperforms traditional indexes in a high-updating environment. In ongoing work, we would explore processing time series with noise or uncertainty and developing index that support temporal and window queries.

## References

1. Saltenis, S., Jensen, C., Leutenegger, S., Lopez., M.: Indexing the position of continuously moving objects. *Proceedings of ACM SIGMOD Conference* (2000) 261–272
2. Tao, Y., Papadias, D., Sun, J.: The TPR\*-Tree: An optimized spatio-temporal access method for predictive queries. *Proceedings of the 29th International Conference on Very Large Databases(VLDB)* (2003) 790–802
3. Tao, Y., Faloutsos, C., Papadias, D., Liu, B.: Prediction and indexing of moving objects with unknown motion patterns. *Proceedings of the SIGMOD* (2004) 611–622
4. Kollios, G., Gunopulos, D., Tsotras, V.J.: On indexing mobile objects. (1999) 261–272
5. Tayeb, J., Ulusoy, O., Wolfson., O.: A Quadtree-based dynamic attribute indexing method. *The Computer Journal* (1998) 185–200
6. Agarwal, P.K., Arge, L., Erickson, J.: Indexing moving points. (2000) 175–186
7. Kwon, D., Lee, S.J., Lee, S.: Indexing the current positions of moving objects using the lazy update R-tree. *The 3rd International Conference on Mobile Data Management* (2002)
8. Lee, M.L., Hsu, W., Jensen, C.S., Cui, B., Teo, K.L.: Supporting frequent updates in R-trees: A bottom-up approach. *Proceedings of the 29th International Conference on Very Large Databases(VLDB)* (2003) 608–620
9. Box, G.E., Jenkins, G.M., Reinsel, G.C.: *Time Series Analysis Forecasting and Control*. Englewood Cliffs, N.J.: Prentice Hall (2004)
10. Tran, N., Reed, D.A.: Arima time series modeling and forecasting for adaptive I/O prefetching. *Proceedings of the 2001 International Conference on Supercomputing* (2001) 473–485