Incremental View-Based Analysis of Stock Market Data Streams

Andreas Behrend Christian Dorau Rainer Manthey Gereon Schüeller
Institute of Computer Science III, University of Bonn
Römerstr. 164, D-53117 Bonn, Germany
{behrend,dorau,manthey,schuelle}@cs.uni-bonn.de

ABSTRACT

In this paper we show the usefulness and feasibility of applying conventional SQL queries for analyzing a wide spectrum of data streams. As application area we have chosen the analysis of stock market data, mainly because this kind of application exhibits sufficiently many of those characteristics for which relational query technology can be considered a valuable instrument in a stream context. The resulting TInTo system is a tool for computing so-called technical indicators, numerical values calculated from a certain kind of stock market data, characterizing the development of stock prices over a given time period. Update propagation is used for the incremental recomputation of indicator views defined over a stream of continuously changing price data.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Design, Performance

Keywords

Data Streams, Update Propagation, Deductive Databases, Sensor Data

1. INTRODUCTION

After more than a decade of research on data stream management, it is widely believed that conventional relational database systems are not well-suited for dynamically processing continuous queries [5, 19, 14]. Therefore, various SQL extensions [11, 21, 2] and stream processing engines have been proposed [1, 3, 4, 18] some of them even designed as full-fledged commercial products (e.g, StreamBase [19]). We believe, however, that even conventional SQL queries can be efficiently employed for analyzing a wide spectrum of data streams [15]. In particular, we think that the application of incremental update propagation considerably improves the efficiency of computing answers to continuous queries.

Update propagation is not a new research topic but has been intensively studied for many years mainly in the context of integrity checking and materialized views maintenance [12]. The application of update propagation for analyzing data streams, however, has not attracted much attention so far. In order to investigate this issue, we have chosen the analysis of stock market data as application scenario. Here, so-called technical indicators are a matter of particular interest. Technical indicators can be expressed as continuous queries (for example in SQL) of considerable complexity, to be evaluated over rapidly changing price data. A technical indicator is basically a mapping from sets of stock price data to numerical values which can be interpreted as buy or sell signal for the respective stock. As these (potentially recursive) indicators are general aggregation functions, they are quite representative for a broad class of stream analysis problems.

For managing technical indicators, we have developed the technical investor tool TInTo [6] where indicators are defined as SQL views. These views are continuously evaluated over a stream of price data pulled from the source `http://finance.yahoo.com` with a frequency of up to one second. Internally, the continuous recomputation of indicator values is carried out by using update propagation. The three main questions addressed by TInTo are: What is the performance gain when applying incremental techniques to the computation of indicator values? Is the resulting TInTo system capable of handling realistic application scenarios involving high-frequency data streams? To which extent are the performance results applicable to other stream-based scenarios?

The aim of this paper is twofold: On the one hand, we will present TInTo as an interesting database application providing insight into the feasibility and flexibility of applying conventional SQL techniques to the view-based analysis of data streams. On the other hand, we will present performance results showing the dramatic impact of using update propagation for evaluating complex aggregate functions over streams. This performance gain fully scales with the size of sliding window and allows to considerably increase the pulling frequency. The main result of this paper supports the claim that the incremental evaluation of SQL views provides a suitable approach for analyzing a wide spectrum of data stream applications. This paper is organized as follows: In Section 2, the main ideas of technical analysis for explaining price movements are presented. Afterwards, the application of SQL views for defining technical indicators is discussed. In Section 4 the technical indicator tool TInTo is presented. The approach for incrementally computing indicators is discussed in Section 5. Performance results for the incremental processing of continuous queries are given in Section 6. Finally, the paper ends with some concluding remarks in Section 7.

2. TECHNICAL ANALYSIS OF STOCKS

Technical analysis is concerned with the prediction of future developments of stock market prices [17]. In contrast to fundamental analysis, it is solely based on the trading history while ignoring the nature of the company or commodity in question. There are several schools of technical analysis and the application of technical indicators is just one of them. In principle, indicators are functions applied to the price history of a certain stock and a point in time. A technical analyst is usually interested in the change of indicator values over a certain time period in order to automatically derive buy or sell signals for stocks. The stream of time-stamped price data is an ordered append-only stream provided by a stock market such as the New York stock exchange. In Figure 1, a typical stream of financial data is presented where the traded volume and the price data of Microsoft is given for the early April 11th in 2008.

Symbol	Price	Volume	Time
...	...	...	...
MSFT	28,01	498	2008-04-11 07:30:21
MSFT	28,34	1834	2008-04-11 07:30:23
MSFT	27,89	650	2008-04-11 07:30:25
...	...	...	...

Figure 1: Stream of Intraday Price Data

The volume data already indicates that the given prices represent average values for time periods of 2 seconds each. In general, indicator computation is based on a division of a stock's price history into consecutive time intervals t_i of equal length. Interval length is user-defined and usually ranges from 1 second to as long as one year. The most common interval value is the typical price which is the mean of the highest, lowest, and closing price of stock s within t_i:

$$\mathrm{TP}(s,t_i) := \frac{\mathrm{high}(s,t_i) + \mathrm{low}(s,t_i) + \mathrm{close}(s,t_i)}{3}$$

We generally assume a fixed interval partitioning where the current point in time lies within an interval t_c, all intervals t_k with $k < c$ contain historical price values and intervals t_l with $l > c$ cover future prices. Note that price data with respect to the current interval t_c is incomplete such that every value based on this data is preliminary. Usually, the actually unknown closing price of t_c is set to the latest price value of the given stream and reset every time a new value is recorded until the end of interval t_c has been reached. Technical indicators are usually applied to the typical price of each interval but also to other characteristic interval values such as highest and closing prices. Due to the price changes for the current time interval t_c, a corresponding indicator value is preliminary and changes, too. An indicator value for historical time intervals, however, remains unchanged.

Technical analysis relies on the assumption that stock prices do not move arbitrarily, but in an up, down, or sideways trend. Consequently, technical indicators have been developed for determining various trend properties and are characterized as trend followers, trend determiners, oscillators and volatility indicators. Because of space limitations we will not consider all indicator types in detail but rather discuss trend followers and oscillators exemplarily. Trend followers are indicators for signalling the principal direction of price movements. A typical and very simple trend follower is the SMA (simple moving average of the typical price) which is defined as follows:

$$\mathrm{SMA}_n(s,t_i) := \frac{\sum_{k=0}^{n-1} \mathrm{TP}(s,t_{i-k})}{n}$$

$\mathrm{SMA}_n(\mathrm{s},\mathrm{t}_i)$ represents the unweighted mean of the typical stock price of s for n consecutive time periods of equal length. The parameter n is provided by the user and usually ranges between 10 and 200. Moving averages such as the SMA are used to smooth out short-term fluctuations, thus highlighting longer-term trends or cycles in the underlying price history. When the stock price rises above the current SMA value, this can be interpreted as the beginning of a positive price trend and, thus, may serve as a buy signal.

The SMA is a very slowly moving indicator where older and newer price data are considered equally important. A possible improvement is to consider an exponential smoothing instead where a weighted mean of the typical stock price is determined. The corresponding indicator EMA (exponential moving average) is generally defined as

$$\begin{aligned}\mathrm{EMA}_n(s,t_i) &:= \mathrm{EMA}_n(s,t_{i-1}) + \mathrm{EW}_n(s,t_i)\\ \mathrm{EW}_n(s,t_i) &:= \tfrac{2}{n+1}\cdot(\mathrm{TP}(s,t_i) - \mathrm{EMA}_n(s,t_{i-1}))\end{aligned}$$

where the weighting factor n is again provided by the user and usually takes the values 26,12 or 9. The EMA indicator is applied quite similar to the SMA. Because of its recursive definition, however, its computation becomes more complex and the question arises how it can be realized in an SQL context

Oscillators are another type of indicators whose values oscillate between upper and lower bounds signaling extreme price values. They usually don't show the direction of a trend but rather the momentum at which the prices move. The following commodity channel index (CCI) is a very popular indicator of this type which typically oscillates above and below a zero line:

$$\mathrm{CCI}_n(s,t_i) := \frac{\mathrm{TP}(s,t_i) - \mathrm{SMA}_n(s,t_i)}{\frac{0.015}{n}\cdot\sum_{k=0}^{n-1}|\mathrm{TP}(s,t_i) - \mathrm{SMA}_n(s,t_{i-k})|}$$

The CCI is often used for detecting divergences from price trends as an overbought or oversold indicator. Readings above +100 imply an overbought condition, while readings below -100 imply an oversold situation. This explains the factor 0.015 which is only used to normalize most of the CCI values to the range of -100 to +100. In a similar way various other indicators have been proposed which are widely used in practice. As in the case of CCI, indicator definitions may be based on other ones, forming a kind of indicator hierarchy. In this way, it is even possible to combine the positive effects of different indicator types, e.g., by connecting a trend determination with an overbought/oversold signal. To which extent technical indicators are meaningful for predicting future price developments, however, remains quite questionable and no statistical significant forecast power has been proved so far. Nevertheless, technical indicators represent a quite general way of analyzing a stream of price data and are provided by most financial trading systems.

3. VIEW-BASED INDICATORS

Although there are various commercial implementations of technical indicators, technical analysts are often interested in building their own indicator system in order to develop a personalized and unique trading strategy. To this end, the appropriateness of new parameter values are investigated and new indicators are invented in order to improve forecast power. A possible way of achieving a flexible and extendible trading system is to implement indicators using a declarative language such as SQL.

The advantage of view-based indicators is that the underlying definition can be easily recovered and modified while new indicators can be simply defined in form of view hierarchies. In addition, the application of SQL views allows the efficient computation of indicator values directly within the database where the portfolio and price data is usually stored and thus avoiding the well-known impedance mismatch. On the other hand, the set-oriented computation of SQL queries is not well-suited for all types of indicators. Another drawback of using SQL is the limited expressiveness of most implemented dialects which cause problems for example when considering recursive indicators such as the above mentioned EMA.

We assume price data to be given in a base table as in Figure 1. According to a chosen time granularity (e.g., 1 minute, 10 minutes, 1 day, or 1 week), a (possibly materialized) view with the schema `prices{[ID,NR,OPEN,HIGH,LOW,CLOSE,TP,Vol]}` is used which summarizes all relevant data with respect to a stock identified by the value of *ID*. This includes for each resulting time interval t_i the opening, closing, highest, lowest and typical price values within t_i as well as the traded volume. The chronological position of an interval is represented by the numerical value of the attribute *NR*. The view `prices` serves as the basic reference for every indicator. However, it seems to be worthwhile to compute indicator values only for the stock currently under consideration. Additionally, almost every indicator definition employs further user-defined parameters such as the number of time intervals n. In order to provide the value n and the stock ID within a view definition, two auxiliary functions `getN()` and `getID()` are employed returning the corresponding values currently set by the user. As an example for their application, consider the following view for defining the SMA

```
CREATE VIEW v_SMA AS
SELECT P1.ID, P1.NR, avg(P2.TP) AS SMAV
FROM prices AS P1 JOIN prices AS P2 ON P1.ID=P2.ID
WHERE P1.ID=getID() AND P2.NR<=P1.NR And
      P2.NR>(P1.NR-getN())
GROUP BY P1.ID, P1.NR ORDER BY P1.NR DESC
```

The SMA value for each time period *NR* is derived by the *SMAV* attribute using SQL's aggregate function `avg`. To this end, a self-join is computed which joins all tuples of `prices` to the n tuples representing previous time intervals. The value of n currently set is returned by the function `getN()` which becomes re-evaluated every time the view is applied. All common indicator definitions can be declaratively specified as SQL views in the way presented above. In Section 2, various indicator types have been introduced but these categories are not really meaningful with respect to their computational complexity. In order to characterize the complexity of view definitions necessary for implementing a given indicator, the following type categories are used:

- *Aggregate-free indicators* are the most simple form of indicators where for each time interval t_i the indicator value can be directly determined using the opening, closing, highest, lowest and typical price value within t_i as well as the trading volume.
- *Aggregate indicators* require the application of an aggregate function to the given price data. We additionally distinguish simple aggregation (as is the case for the SMA) from multi-level aggregation where an aggregate function must be applied to an already aggregated value. The latter indicator type often requires the definition of a view hierarchy for obtaining intermediate aggregation values (e.g. the CCI).
- *Recursive indicators* require a self-reference within a view-based definition (e.g. the EMA). These indicators are particularity demanding to compute and pose problems to systems providing no or only a limited form of recursive views.

The oscillator ADO [17] is an example of a simple aggregation-free indicator but will not be discussed in this paper. The above-mentioned CCI, however, obviously requires multi-level aggregation as it is based on SMA values. For its computation, we first derive tuples containing the SMA values and the typical prices of a stock for each time period *NR*:

```
CREATE VIEW v_CCIaux AS
SELECT T.ID, T.NR, T.TP, V.SMAV
FROM prices AS T JOIN v_SMA AS V
     ON (T.ID=V.ID) AND (T.NR = V.NR);
```

A self-join of `v_CCIaux` can be employed to compute the CCI values with respect to the current number of time intervals n returned by the function `getN()`

```
CREATE VIEW v_CCI AS
SELECT V1.ID, V1.NR,(V1.TP-V1.SMAV)/
      (0.015*AVG(ABS(V2.TP-V2.SMAV))) AS CCIV
FROM v_CCIaux AS V1 JOIN v_CCIaux AS V2
     ON V1.ID = V2.ID
WHERE V1.ID=getID() AND V2.N<=V1.N And
      V2.N>(V1.N-getN())
GROUP BY V1.ID, V1.NR, V1.TP,V1.SMAV
ORDER BY V1.NR DESC;
```

In spite of the elaborate CCI definition, its view-based implementation remains rather simple and can be easily modified, for example, if one wants to use normalization factors different from 0.015. Recursive indicators are especially problematic as it is not clear how to implement them in systems which do not support recursive views. Many commercial products such as MS Access or MySQL, however, extend the syntax of the UPDATE command by a possible ORDER BY-clause allowing to specify the order in which a set of tuples is to be updated. This facilitates the simulation of linear recursion and thus, the implementation of almost every recursive indicator. In fact, the authors are unaware of any technical indicator used in practise which is defined using nonlinear recursion.

As an example, consider the recursive indicator EMA from Section 2 where the value $\text{EMA}_n(s, t_i)$ depends on the indicator value $\text{EMA}_n(s, t_{i-1})$ for the previous time period t_{i-1}. Instead of a view, the table t_EMA with the schema t_EMA{[ID,NR,EMAV] is used for computing EMA values by two modifications. First, the attribute value of *EMAV* is initialized with the typical price for each time period *NR*:

```
INSERT INTO t_EMA (ID,NR,EMAV)
SELECT ID, NR, TP AS EMAV FROM prices;
```

Afterwards, an ordered update statement can be employed to successively compute EMA values starting from the earliest time period:

```
UPDATE t_EMA JOIN t_EMA AS EO
ON (t_EMA.ID=EO.ID) AND (t_EMA.NR-1=EO.NR)
SET t_EMA.EMAV = (EO.EMAV + (2/(getN()+1))*
                 (t_EMA.EMAV-EO.EMAV))
WHERE t_EMA.ID=getID()
ORDER BY t_EMA.ID, t_EMA.NR
```

The 'ORDER BY'-clause leads to a sequential update of the tuples in t_EMA starting with the one having the smallest value in *NR*. Another peculiarity is the application of the join operator within the update statement allowing to access attribute values of a different table or view. In our example, a self-join is employed that joins two tuples with consecutive time periods. Consequently, the EMA value of the previous time period *NR-1* can be accessed while updating the EMA value with the time period *NR*. Note that including a join into an update statement violates the SQL standard despite of being widely used in various commercial systems such as SQL Server, DB2, MySQL, and MS Access. We have used the DB2 syntax for expressing a table update with join although an equivalent update statement can be formulated in standard SQL by using subqueries for each attribute of t_EMA. The effects of the 'ORDER BY'-clause, however, cannot be realized in standard SQL.

A possible disadvantage of using a table instead of a view for implementing the recursive indicator EMA is the additional overhead for keeping the table t_EMA up-to-date. In Section 5, however, we will show how indicator views can be incrementally updated which will require their materialization anyway. The additional costs for keeping the materialized views and such tables up-to-date turned out to be quite similar and can be neglected in view of the immense performance gain resulting from their incremental maintenance.

4. THE TINTO SYSTEM

TInTo is an experimental system aiming at demonstrating the usefulness and feasibility of applying conventional SQL queries for implementing technical indicators [6]. The acronym TInTo is used as an abbreviation for Technical Investor Tool indicating its task as automatic trading system based on indicator signals. The system has been implemented by several Master Theses at the University of Bonn, unfortunately documented in German only (e.g.[16, 20]).

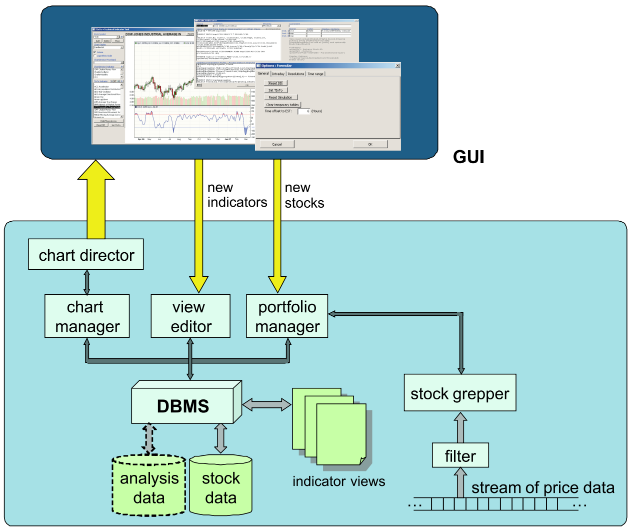

Figure 2: Architecture of TInTo

Initially, TInTo has been a Visual Basic (VBA) application based on MS Access only, but is currently being reprogrammed as a web-based application using Java and Oracle. TInTo manages a user-defined portfolio of indices, commodities, and stocks whose historical as well as intraday quotations are provided by http://finance.yahoo.com for stock markets worldwide. As a frontend it uses the shareware visualizer ChartDirector [10], a tool supplying a VBA library of well-established methods for drawing financial charts. Even though ChartDirector comes along with a wide spectrum of built-in technical indicators (computed by VBA functions) we extended the tool by a means to specify arbitrary indicators as predefined SQL queries (i.e., as views), evaluated directly over the underlying database. The values of these query-based indicators are visualized by ChartDirector in the same way as those computed by the tool's own indicator functions. By extending the tool with a simple SQL view editor we can offer a system for specifying new and modifying existing indicator definitions in an extensible manner. At present we experiment with some 30 view-based indicators making use of various SQL features.

An overview of TInTo's architecture is given in Figure 2. The view editor allows to freely define indicator views which may directly access the table of timestamped prices and/or other indicator views already defined. Some attributes, however, are predefined for controlling the visualization of view data by ChartDirector. This includes threshold values for indicators such as the CCI signaling sell or buy situations for the sock under consideration.

Even though a considerable degree of analysis is reachable this way, hardly any streaming is involved yet, unless one already considers e.g. the sequence of daily closing prices of stocks as a very low frequency "stream". The crucial step towards proper stream management in TInTo consisted in the addition of a VBA script automatically downloading a record of new price and volume values per stock in the portfolio at regular intervals and appending the downloaded data to those already present in the database. The frequency of download can be freely configured by the TInTo user and may even be set as low as 1 second. The software component thus realized - which we will call Stock Grepper in the following - generates a data stream pulled from a permanent data source on demand as long as the script is active.

The Stock Grepper uses a Filter component to remove erroneous trading data from the generated stream, e.g., price tuples with a sales volume set to zero. The remaining downloads are appended to a price table quite similar to the one depicted in Figure 1. Afterwards, an action query is employed to determine or update the synopsis table `prices` with aggregated price data according to the current time period setting chosen by the user. The Chart Manager prepares the price as well as indicator data of selected stocks and indicator functions for their visualization by ChartDirector. It additionally initiates the redrawing of charts as soon as new price data arrives. The Portfolio Manager allows to search for diverse stocks and commodities whose market quotes are provided by YAHOO. For each portfolio selection of a user, it additionally ensures all quotes to be loaded which have been missed during an offline phase.

5. UPDATING INDICATORS

Update propagation has been studied mainly in the context of integrity checking and materialized views maintenance [12]. The key idea is to transform each SQL view already at schema design time to a so-called *delta view*, a specialized version of the view referring to changes in the underlying tables, only. The original view definitions are employed only once for materializing their initial answers while the specialized versions are used afterwards for continuously updating the materialized results. Under the assumption that a great portion of the materialized view content remains unchanged, the application of delta views may considerably enhance the efficiency of view maintenance. In our group, we contributed to the development of delta techniques e.g. within the international IDEA project during the 1990s (cf. [13]). We adopted this idea for the TInTo system, using delta-view techniques for a synchronized update of indicator values. To this end, update statements are applied instead of the original indicator views for incrementally maintaining the materialized indicator values. In principle, these update statements can be automatically compiled from the original views. However, we do not have a full-fledged delta compiler for arbitrary SQL views yet, therefore we are performing our experiments with hand-compiled delta views for the time being. In the following, we will discuss how to derive specialized update statements for selected indicator views, thus illustrating the general approach.

In Section 3 we introduced the materialized view `prices` which serves as the basic reference for every indicator in TInTo. Having a primary stream of price data, the aggregated values for each time interval in `prices` represent a synopsis of this stream which has to be continuously updated. For instance, consider the following snapshot of `prices`

ID	NR	...	TP	Vol
1	250	...	25,23	1200
1	249	...	23,44	260
1	248	...	21,22	2300
1	247	...	22,55	300
...	...	...	...	...

comprising 250 time intervals for the stock with *ID*= 1. The tuple with *NR*= 250 represents the latest time interval t_c in which the current point in time lies. A new tuple coming from the primary stream of price data leads to two possible modifications of `prices`. If the timestamp of this price still falls into the current time interval t_c, the highest, lowest, closing and typical price values of the tuple with *NR*= 250 are modified accordingly. Otherwise, a new time interval has to be considered, leading to an insertion of a corresponding tuple with *NR*= 251.

Indicators based on summarized price data represent predefined continuous queries which have to be reevaluated as soon as `prices` changes. As an example, consider again the indicator SMA

$$\mathrm{SMA}_n(s,t_i) := \frac{\sum_{k=0}^{n-1} \mathrm{TP}(s,t_{i-k})}{n}$$

and the corresponding view `v_SMA` for computing SMA values based on the typical prices for each time interval in table `prices`. To this end, a self-join has been computed joining all intervals from the first version P1 of `prices` to n previous intervals from the second version P2 of `prices`. Under the assumption that the user-defined value n of the formula is currently set to 3, the following temporary cross product is internally generated before the average function can be applied:

P1.ID	P1.NR	P2.NR	P2.TP	...
1	250	250	25,23	...
1	250	249	23,44	...
1	250	248	21,22	...
1	249	249	23,44	...
1	249	248	21,22	...
...	...	...	...	...

As mentioned above, a new price tuple leads either to a modification of or insertion into `prices`. Suppose the tuple representing the characteristics of t_c in `prices` must be modified due to new data coming from the primary data stream. The corresponding entry is recorded in the following delta-table `u_prices` for updates:

ID	NR	...	TP	Vol
1	250	...	26,11	230

In [7], the authors described the Magic Update propagation method where Magic Sets [8] is employed to select the relevant part of temporary cross products needed for the subsequent incremental maintenance of the resulting view. In our case, this involves the first group of tuples with *P1.NR*= 250. These values are determined by the following auxiliary view `v_mri_SMA`

```
CREATE VIEW v_mri_SMA AS
SELECT P2.*
FROM u_prices AS P1 JOIN prices AS P2
      ON P1.ID = P2.ID
WHERE P1.ID=getID() AND P2.NR<=P1.NR And
      P2.NR>(P1.NR-getN())
GROUP BY P1.ID, P1.NR ORDER BY P1.NR DESC
```

which comprises the tuples of the most recent interval, and thus the first three tuples of the temporary cross product above. To this end, the original view definition `v_SMA` is modified using `u_prices` instead of `prices` as first join partner in order to select only those tuples from the original join which are affected by the modification in `u_prices`. The view `v_mri_SMA` can now be used to update the materialized SMA values stored in the table `t_SMA`. For incrementally updating average values, the following formula can be applied

$$\text{AVG}_{\text{new}} := \text{AVG}_{old} - \frac{x_{old}}{\text{n}} + \frac{x_{new}}{\text{n}}$$

if the value x_{old} is updated to x_{new} for an average computation of n values. Under the assumption, that `prices` still provides the old typical price values, the following update statement can be employed for incrementally deriving the new SMA value of t_c:

```
UPDATE t_SMA SET
SMA = SMA - (SELECT TP/getN() FROM v_mri_SMA
             WHERE v_mri_SMA.NR=NR)
          + (SELECT TP/getN() FROM u_prices
             WHERE u_prices.NR=NR)
WHERE ID=getID() AND
      NR=(SELECT P.NR FROM u_prices)
```

Its application to our sample instance `prices` would result in a single modification of the materialized SMA values in `t_SMA` where the old SMA value 23,30 of the 250th time interval is updated to 23,59:

ID	NR	SMAV
1	250	23,59
1	249	22,40
...	...	...

In an analogous way, the insertion of a new SMA value can be incrementally determined if a new time interval t_c^{new} with $NR = 251$ has started. Note that the higher the value of n is set by the user, the faster this incremental approach performs in comparison to a complete recomputation. The significant performance gain shown in Section 6, however, additionally results from the high number of historical SMA values remaining unchanged.

In a similar way, the values of the CCI indicator can be incrementally maintained. To this end, the auxiliary view `v_CCIaux` as well as `v_CCI` are materialized and corresponding modifications are separately stored using the delta tables `u_CCIaux` and `u_CCI`, respectively. Because of the multi-level aggregation, however, it is necessary to determine the delta tables before the actual updates are applied to the materialized views `t_CCIaux` and `t_CCI`. The reason for this are references to old values (such as x_{old} from above) which must not be overwritten by an update statement before the next level of views within the given hierarchy is completely processed (a "stratification" phenomenon). Consequently, an insertion statement is used this time for determining the delta relation `u_SMA` instead of directly applying the corresponding update statement from above:

```
INSERT INTO u_SMA SELECT New.ID,New.NR,
      I.SMAV-(Old.TP/getN())+(New.TP/getN())
FROM u_prices AS New JOIN t_SMA AS I JOIN
     v_mri_SMA AS Old ON
     (New.ID=I.ID) AND (New.ID=Old.ID) AND
     (New.NR=I.NR) AND (New.NR=Old.NR);
```

Based on `u_SMA`, the delta relation `u_CCIaux` can be determined using

```
INSERT INTO u_CCIaux
SELECT T.ID, T.NR, T.TP, V.SMAV
FROM u_prices AS T JOIN v_SMA AS V
     ON (T.ID=V.ID) AND (T.NR = V.NR);
```

before it is actually applied to update `t_SMA`. The update in delta table `u_CCIaux` is again not directly applied to `t_CCIaux` but first used to determine the next-level delta table `u_CCI` containing changes of the materialized CCI values. Afterwards, all necessary delta tables are determined and the remaining tables `t_CCIaux` as well as `t_CCI` can be updated using these deltas. Note that recursive indicators like EMA can be incrementally maintained in this way, too. The computation of EMA updates is even less expensive than the one for determining CCI updates because of its simple linear recursion based on typical prices.

In the same way, all other views employed in our system are systematically transformed into specialized update statements. The transformations outlined above follow general principles (outlined by the authors in [7]) and don't depend on the specific characteristics of the underlying application. That is, all update statements are systematically derived from the original view definitions, even though it is sometimes possible to find even simpler solutions for incremental maintenance of certain indicators if taking their semantics into account. The reason for preferring automated compilation of delta-views, however, is our intention to develop a general approach for the view-based analysis of data streams where automatically derived delta-based statements are employed for the incremental evaluation of continuous queries. The feasibility of our approach is shown in the following section where performance results are presented.

6. PERFORMANCE RESULTS

All indicator queries are based on a sliding window defined by the time span for which the user wants to see indicator values. In TInTo, a user determines the window size by setting the time range attribute which may take values from 1 day to 20 years. The price and indicator values of all time intervals falling into the chosen time range are visualized by ChartDirector. As soon as a new current time interval t_c^{new} is considered, the oldest one and its corresponding indicator values are removed from the list of tuples to be displayed, and a new entry is made for t_c^{new}. Usually, the window size and interval length are chosen by the user in meaningful and balanced way such that always a considerable number of intervals fall into the chosen time range of the sliding window while only one interval is continuously updated.

In Figure 3, a comparison of the evaluation times of a non-incremental (naive) implementation of ADO and EMA based on non-incremental views vs. an implementation based on delta views is shown. While the required time of non-incremental computation significantly increases when more tuples are considered, the time needed for delta view evaluation remains almost unchanged. The performance gain becomes even more dramatic if more complex indicators such as the CCI or SMA are considered. This is illustrated in Figure 4 where a linear scale is no longer sufficient to illustrate the performance difference in a meaningful way and a logarithmic scale has to be used.

Note that up to a window size of 30 tuples, the application of delta views achieves no performance gain in comparison to the naive one. This is due to the computational overhead

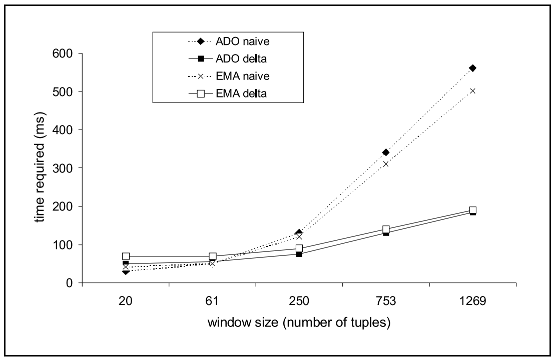

Figure 3: ADO/EMA computation

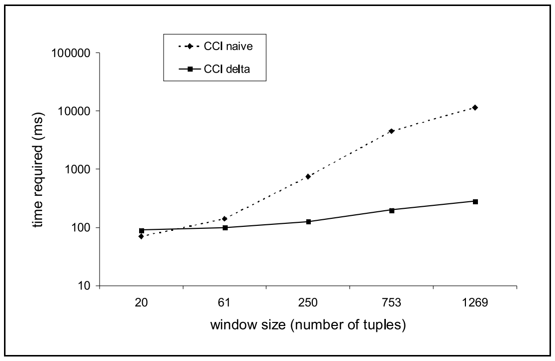

Figure 4: CCI computation

resulting from more complex query specifications and the maintenance of materialized data. It is remarkable that the performance gain fully scales with the sliding window size as well as indicator complexity. That is, with an increasing size of the underlying sliding window the performance gain by the application of update propagation increases as well.

Increase in efficiency of continuous query evaluation obtained so far in many cases encourages us to continue along this line. However, translating transformation methods defined mostly in a Datalog context in the literature on SQL is a non-trivial task (not yet mastered by relational DBMS vendors even for integrity and view materialization purposes), so that we did not yet invest too much effort in delta compiler construction.

7. CONCLUSION

We have presented the technical investor tool TInTo which allows the computation of technical indicators over a stream of stock prices using SQL views. In order to perform a synchronized update of indicator values, the TInTo system employs delta-views over the high frequent stream of stock market prices. Compared with the complete re-evaluation of indicator views, this incremental approach led to a remarkable performance gain allowing to recompute multi-level aggregates within microseconds. For quite a wide range of realistic trading strategies even a limited system like TInTo has proven to be sufficiently powerful to master the data size and stream frequency needed to perform the required analytical tasks in SQL. This encourages us to believe that traditional relational database techniques are indeed suited for analyzing a wide spectrum of data streams.

8. REFERENCES

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom: *STREAM: The Stanford Stream Data Manager.* SIGMOD 2003: 665.

[2] A. Arasu, S. Babu, and J. Widom: *The CQL continuous query language: semantic foundations and query execution.* VLDB J. 15(2): 121-142.

[3] D. J. Abadi et al.: *Aurora: A Data Stream Management System.* SIGMOD 2003: 666.

[4] D. J. Abadi, W. Lindner, S. Madden, and J. Schuler: *An Integration Framework for Sensor Networks and Data Stream Management Systems.* VLDB 2004, 1361-1364.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom: *Models and Issues in Data Stream Systems.* In *PODS*, pages 1–16, 2002.

[6] A. Behrend, C. Dorau, and R. Manthey: *TinTO: A Tool for the View-Based Analysis of Streams of Stock Market Data.* DASFAA, 2007: 1110-1114.

[7] A. Behrend, R. Manthey: *Update Propagation in Deductive Databases Using Soft Stratification.* ADBIS 2004: 22-36

[8] F. Bancilhon, D. Maier, Y, Sagiv, and J. D. Ullman: *Magic Sets and Other Strange Ways to Implement Logic Programs.* PODS 1986: 1-15

[9] S. Babu, J. Widom: *Continuous Queries over Data Streams.* SIGMOD Record 30(3): 109-120 (2001)

[10] *Chart Director.* http://www.advsofteng.com (2006)

[11] D. Chatziantoniou, Y. Sotiropoulos: *Stream Variables: A Quick but not Dirty SQL Extension for Continuous Queries.* ICDE Workshops 2007: 19-28

[12] A. Gupta, I. S. Mumick: *Materialized Views: Techniques, Implementations, and Applications.* The MIT Press (1999)

[13] U. Griefahn, T. Lemke, and R. Manthey: *Chimera Prototyping Tool: User Manual.* Technical Report IDEA.DE.22.O.006, ESPRIT Project 6333 (IDEA), 1996

[14] L. Golab, M. T. Özsu: *Issues in data stream management.* SIGMOD Record 32(2): 5-14 (2003)

[15] A. Hoppe, J. Gryz: *Stream Processing in a Relational Database: A Case Study.* IDEAS, 2007: 216-224.

[16] C. Hübel: *TInTo - Ein datenbankgestütztes Werkzeug zur regelbasierten Wertpapieranalyse.* Master Thesis, University of Bonn, 2007.

[17] C. Le Beau, D. Lucas: *Technical Traders Guide to Computer Analysis of the Futures Markets* . Irwin Professional (USA), 1992

[18] S. Madden, M. J. Franklin: *Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data* ICDE 2002: 555-566.

[19] M. Stonebraker, U. Çetintemel: *"One Size Fits All": An Idea Whose Time Has Come and Gone (Abstract).* ICDE, 2005: 2-11.

[20] G. Schüller: *Änderungspropagierung zur regelbasierten Analyse von Finanz-Datenströmen in TInTo* Master Thesis, University of Bonn, 2007.

[21] H. Wang, C. Zaniolo, and C. Luo: *ATLAS: A Small but Complete SQL Extension for Data Mining and Data Streams* VLDB 2003: 1113-1116.