18th International Conference on Information Fusion
Washington, DC - July 6-9, 2015

Air Traffic Monitoring using Datastream Analysis Techniques

Gereon Schueller *, Philip Schmiegelt *, Andreas Behrend †
*Fraunhofer FKIE, Wachtberg, Germany, first.last@fkie.fraunhofer.de
†University of Bonn, Bonn, Germany, behrend@cs.uni-bonn.de

***Abstract*—The "AIrspace Monitoring System" (AIMS) analyzes flight data streams with respect to the occurrence of freely definable complex events considered critical or at least relevant by its users. In contrast to already existing tools which often focus on a single task like flight delay detection, AIMS represents a general approach to a comprehensive analysis of aircraft movements, derived from transponder and/or radar measurements. It has been developed for showing the usefulness and feasibility of applying conventional SQL queries for analyzing rapidly changing sensor data. The key innovative feature of AIMS is that we apply Magic Sets to incremental view maintenance techniques in order to process data streams in a typical real-time scenario.**

I. Introduction

The continuous growth of traffic in air space challenges existing monitoring systems for air traffic control. In fact, there are still plenty of situations where anomalies or critical events are detected too late or remain undetected at all. Many problems are caused by local deviations from flight plans which may induce global effects to aircraft traffic. One effect is a considerable number of close encounters of planes in airspace occurring every day, as well as the violation of no-fly zones.

The "AIrspace Monitoring System" (AIMS) [15] is a prototype of a system for monitoring and analyzing local and global air traffic. The aim of this system is the anomaly detection on individual aircrafts movements.

Based on that, a global analysis is supported (e.g., critical encounters, zones with high flight density, airport jams). To this end, consistent tracks of individual planes have to be derived and complex event occurrences within these track data have to be found in nearly real-time. Currently, the system is able to monitor the complete German airspace, i.e. there are measurements every 4 seconds with up to 2000 flights in peak times.

The key innovative feature of AIMS is the use of continuously evaluated SQL queries which are automatically materialized and incrementally evaluated by the underlying DBS. Using SQL queries as executable specifications has the advantage of being able to easily extend the system by additional criteria without having to re-program large amounts of code. In order to continuously re-evaluate the respective queries, data stream management systems (DSMSs) such as STREAM [1] or Aurora [3] could be used. However, DSMSs do not provide all capabilities that are needed for a reliable in-depth analysis, like recovery control, multiuser access and processing of historical as well as static context data. This is why our general research aim is the development of incremental DBMS-based methods for real-time gathering and monitoring of streams of track data.

With respect to data stream management, the following research questions are addressed by AIMS:

- Which type of continuous queries can be efficiently evaluated in an incremental way using a DBMS?
- Up to which frequency/volume is it feasible to use an incremental approach within a relational DBMS for evaluating continuous queries?
- How can an efficient and transparent caching approach be realized in this stream context?

In this article, we show how a commercial DBMS in combination with intelligent rule-rewriting can be used to process a stream of temporal geospatial data. The proposed approach provides insights useful for the implementation and optimization of related applications where geospatial or sensor data streams have to be analyzed and monitored. We also applied this method to the task of probabilistic tracking [6], [17], where radar plots have to be assigned to tracks, showing the general feasibility of our methods.

II. The AIMS System

A graphical representation of the architecture of AIMS is shown in Fig. 1. AIMS consists of three main components:

1) A feeder component which takes a geospatial data stream as input and periodically pushes its data into the database. This track feeder also continuously activates the re-evaluation of the anomaly detection views.
2) An Oracle server which stores the stream data in regular intervals (every 4 seconds or less) and performs the continuous evaluation of the user-defined anomaly detection views.
3) A client program which contains a graphical user interface and a cache component. The GUI shows the positions of tracked aircrafts on an OpenStreetMap. An in-memory database works as a cache and stores the results of selected queries on the client side. This

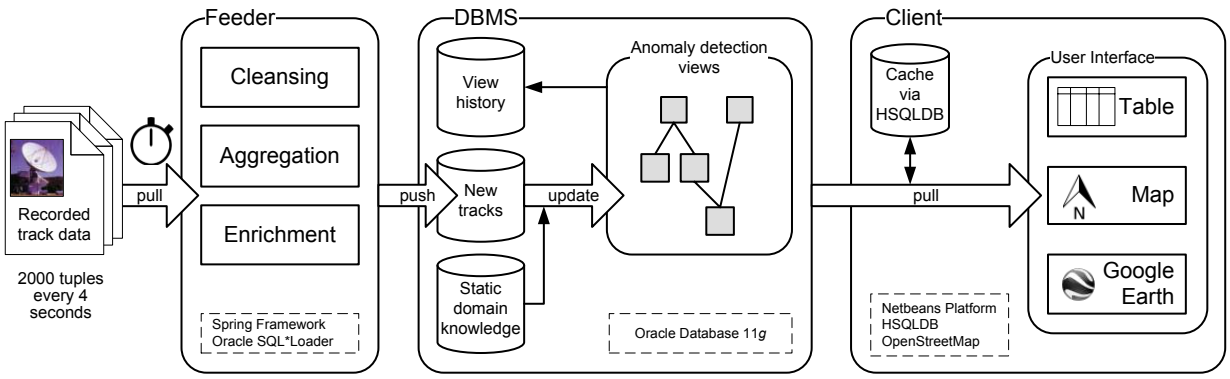

Fig. 1: The architecture of AIMS

database forms the basis for a time-shift and video recorder functionality.

The system works on pre-processed tracks which are computed from radar data and transponder signals. The tracking algorithm calculates probabilistic values for a time-series of inaccurate sensor data quantifying the likelihood that a measurement belongs to the track of a moving object or not. The resulting tracks and transponder signals are then merged into a stream of highly accurate aircraft positions. In this way, missed radar measurements, misdetections (clutter) and the assignment problem if multiple objects are in the field of view are handled.

The resulting stream of timestamped position and velocity data is periodically pushed into the DBS using the feeder component. Every 4 seconds, new track data is provided and stored in a 'delta table' containing just the most recent track data. Its former content is moved to a history table such that the complete track of each flight is recorded. A sample output of our tracking algorithm may look as follows:

time	id	lon	lat	alt	vel	...
19:20:43	34	51.12	7.05	4534	300.4	...
19:20:44	35	50.98	6.34	2324	240.3	...
...	...	...	...	...	...	...

Questions to be continuously answered using SQL queries are, e.g.:

- Which aircrafts are currently landing? Which aircrafts are airborne?
- Which aircrafts approach a bad weather zone or are over a certain region?
- Which aircrafts approach each other critically?
- Are there critical deviations from flight plans?
- What is the average number of landings for a user-chosen airport?

The corresponding views are stored in and managed by the underlying relational database system. The client of AIMS provides a graphical user interface that allows the user to select anomaly detection views from the database server and to monitor their results. It also displays performance measurements for each periodic query execution. All of the above queries can be executed in less than 2 seconds which is important because it is below the radar refreshment rate of 4 seconds.

Another feature of the client is a "time-slider" functionality for reviewing the recent track history [16]. To this end, the in-memory database system HSQLDB [10] is used to store the most recent continuous query results and thus allows for a rewind to and a replay from an arbitrary time point in the past. For example, the development of critical encounters could be comprehensively analyzed this way. In addition, the cached query results can be further analyzed using refined SQL statements. For example, new criteria not checked in the original analysis process like "display all flights in a region of 30 km around the critical encounter" or "give all tracks that had a critical encounter" could be applied to the obtained query results.

Even though a considerable degree of analysis can be achieved by purely recomputing expressive SQL views in each refreshment cycle, the given stream scenario will sooner or later drastically slow down our system without further optimization. In the following sections, we will explain in more detail how incremental evaluation techniques can be used for an efficient view-based analysis of stream data.

III. View-Based Flight Analysis

Although there are various commercial implementations of flight tracking services (e.g., FlightView [9] or FlightStats [8]), they are often limited to a set of predefined tasks like delay detection or identification of basic flight states such as departing, approaching or cruising. In recent time, some work has been done for the automatical analysis of hazards in Airspace [11], [12], [7]. However, these systems have not been extended to continuous monitoring systems. In order to develop a flexible and extensible monitoring system, we have decided to use SQL views for analyzing track data. The view-based analysis is performed over a stream of track data and almost static domain knowledge

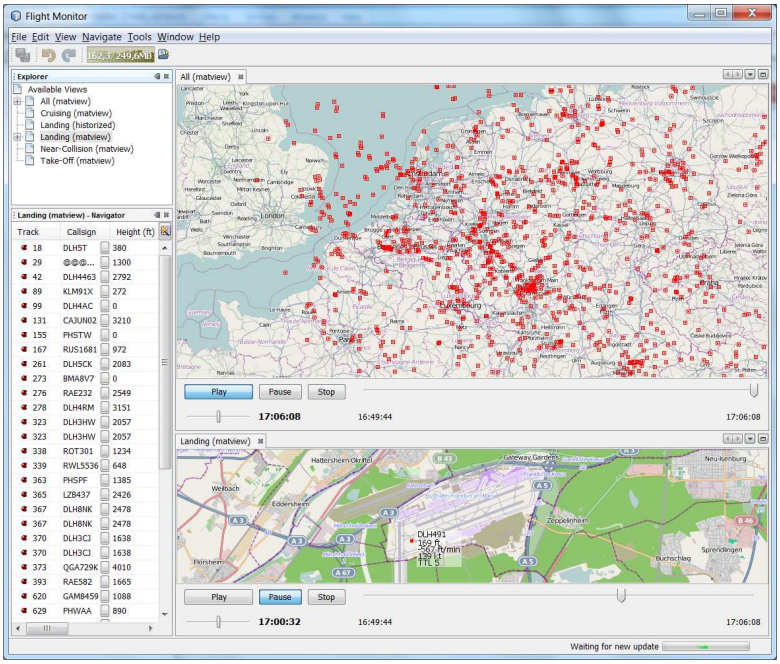

Fig. 2: The GUI of AIMS is programmed in Java using the NetBeans Platform library. It shows the positions of analyzed aircrafts on an OpenStreetMap. This map can be configured in order to show the result of several selected queries at the same time, e.g. all airborne flights and current landings at Frankfurt. Additionally, query results are displayed in tabular form.

about flight plans and airport specifications. The track data contains up to 69 attributes including track id, call sign, timestamp, position, various altitude and velocity measurements as well as a maneuver respectively ACAS-report. The flight plans are given in form of a series of waypoints associated with every flight. The airport data contains IATA/ICAO codes as well as latitude and longitude values.

In the following, we will discuss two examples of anomaly detection, using Datalog expressions rather than SQL queries for the sake of the simplicity. The first example considers landing airplanes whereas the second one returns critical encounters. Afterwards, we combine both queries in order to detect landing planes on a collision course showing the flexibility of our analysis.

a) Landings: Usually, a continuous query is based on rapidly changing track data as well as on fixed domain knowledge. As an example for this combination let us consider landing flights. To this end, the plane must have a negative vertical speed (i.e., it is descending), it has to be below a certain flight level (like 3000 ft), and it must be in the vicinity of an airport, e.g. closer than 20000 ft. These criteria can be expressed in SQL as follows:

$$\begin{aligned} \text{landing(id)} \leftarrow & \text{tracks}(v_{\text{vert}}, \text{level}, \text{pos}_1), \\ \leftarrow & \text{airports}(\text{pos}_2), v_{\text{vert}} < 3000\text{ft/min}, \\ & \text{level} < 3000\text{ft}, ||\text{pos}_1 - \text{pos}_2|| < 20\text{km} \end{aligned} \tag{1}$$

The user-defined function (UDF) `dist` calculates the Euclidean distance between two positions on the globe. The table `airports` stores data about position and names of airports.

This view returns all tracks that can be classified as landings, including those detected a long time ago. In a real-time application, it is typically much more interesting to find all those tracks associated with currently landing flights. A simple solution would be the selection of the respective tracks using the most recent timestamps. However, this would require an additional selection criterion to be added to every view. In order to avoid this costly overhead, we provided the necessary focus in a different way, two tables are used: a delta table for storing the newest flight tracks and a history table for recording older track information. For synchronizing the two tables, a set consisting of the following update rules is employed:

$$\begin{aligned} \Delta^+\text{hist}(\text{track\_id}, \cdots) \leftarrow & \Delta^+\text{tracks}(\text{track\_id}, \cdots) \\ \Delta^+\text{trackTTL}(\text{id}, \text{TTL}) \leftarrow & \text{trackTTL}(\text{id}, \text{TTL}), \\ & \neg\Delta^+\text{tracks}(\text{id}, \_) \\ & \underline{\text{with}}\ \text{TTL} := \text{TTL} + 1 \\ \Delta^+\text{trackTTL}(\text{id}, '15') \leftarrow & \Delta^+\text{tracks}(\text{track\_id}, \_), \\ & \neg\text{trackTTL}(\text{id}, \_) \end{aligned} \tag{2}$$

This updates the delta table and moves its former content to the history table. The table `tracks` maintains all track data arriving in one of the last 15 refreshment cycles by decrementing and resetting a time to live (TTL) attribute. This technique is important because not all tracks are caught with every update, since tracks may be missed or delayed, and a simple deletion could erase still active tracks. The same technique can be applied to other problems arising in air space monitoring, as will be shown in Section IV.

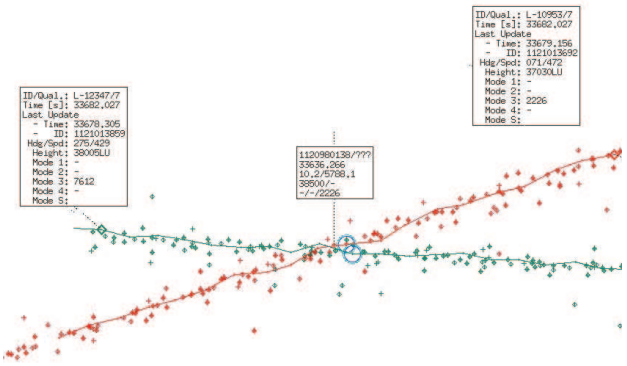

Fig. 3: Angular view of a critical Encounter

The focus on new track information is employed in almost every anomaly detection view we use. However, even this focus on new track data is sometimes not sufficient for avoiding performance problems.

b) Critical Encounters: As an example, consider the view definition for finding critical approaches. Suppose the position and velocity of two planes is given by the vectors $\vec{p_1}$ and $\vec{v_1}$ as well as $\vec{p_2}$ and $\vec{v_2}$, respectively. The two planes are on a potential collision course if the following conditions hold:

1) There are two scalar values λ_1 and λ_2 such that the following equation holds:
$$\vec{p_1} + \lambda_1 \vec{v_1} = \vec{p_2} + \lambda_2 \vec{v_2} \tag{3}$$
2) Both planes fly towards the potential collision point p_c, i. e., $\vec{p_c} - \vec{p_i}$ is parallel but not anti-parallel with respect to $\vec{v_i}$.
3) Both planes would arrive at the collision point simultaneously, i. e.
$$\frac{\text{dist}(\vec{p_1}, \vec{p_c})}{\vec{v_1}} = \frac{\text{dist}(\vec{p_2}, \vec{p_c})}{\vec{v_2}} \tag{4}$$

In practice, however, the above conditions are much too sharp and a general security distance d_c has to be additionally taken into account. This can be achieved by simply modifying conditions 1 and 2 as follows:

1) The distance between the skew lines derived from the plane trajectories is smaller than the critical distance d_c (e.g., $d_c \approx 300$ feet):
$$d = (\vec{p_1} - \vec{p_2}) \cdot \frac{(\vec{v_1} \times \vec{v_2})}{|\vec{v_1} \times \vec{v_2}|} < d_c \tag{5}$$
2) The time interval in which the planes would reach the collision point is smaller than the critical time range t_c (e.g., $t_c \approx 60$ seconds):
$$\left| \frac{\text{dist}(\vec{p_1}, \vec{p_{c1}})}{\vec{v_1}} - \frac{\text{dist}(\vec{p_2}, \vec{p_{c2}})}{\vec{v_2}} \right| < t_c \tag{6}$$
The position vectors $\vec{p}_{c\{1,2\}}$ denote the points on the lines where the common perpendicular crosses.

It is obvious that the calculation of planes on a collision course would consume much time if all planes currently flying were checked against each other. However, there is a simplification that will drastically speed up the computation. The idea is to check only those planes that occupy adjacent quadrants such that a collision detection becomes crucial. To this end, the absolute value of the longitude and latitude difference for the respective planes should be below 1. The reason is the natural speed maximum for common planes (less than 300 m/s) such that collision points which take more than 40 minutes to be reached can be ignored. In fact, during that time the course will typically change anyway such that a potential collision will not be detected later. An unavoidable drawback of this approach is that it will not work near polar regions.

Using the above selection conditions, a rule set for detecting collision courses could be defined as follows:

$$\begin{aligned} \text{encounter(id)} \leftarrow & \Delta\text{track}(\text{id}_1), \text{t}_2(\text{id}_1). \\ \text{t}_2(\text{id}) \leftarrow & \Delta\text{track}(\text{id}_1, \text{la}_1, \text{lo}_1), \\ & \text{id}_1 \neq \text{id}_2, \\ & ||\text{lo}_1 - \text{lo}_2|| < 1, ||\text{la}_1 - \text{la}_2|| < 1, \\ & \text{doCollide} == \text{TRUE} \end{aligned} \tag{7}$$

where the user defined functions `doCollide` returns TRUE if both planes satisfy the criteria specified in equations 5 and 6 from above. In our scenario with up to 2000 flights, this view can be executed in less than 1 second. An example of two flights critically approaching each other is given in Figure 3. It shows the tracks of two civil aircrafts with a slope distance of less than 1000 meters and a vertical distance of less than 300 feet. The critical approach itself is indicated by the two circles in the middle of the picture while the courses of the respective airplanes 30 seconds before and after this event are depicted, too. Note that the curved lines already represent computed tracks which result from interpreting the dotted radar data also provided in the picture. In Figure 4, a different view of a similar event is given where a descending airplane critically approaches a cruising one. Again, the critical approach itself is indicated by the two circles in the middle of this picture.

c) Critical Landings: The client of AIMS allows to freely add new user-defined anomaly detection views which are to be continuously evaluated by the system. For example, a user may combine landing flights and close encounters in order to define a new view for determining critical landings:

$$\text{critLanding(id)} \leftarrow \text{encounter(id)}, \text{landing(id)}. \tag{8}$$

IV. Robust Flight Phase Detection

In principle, we distinguish between the determination of a certain system state from the occurrence of an event. A flight state usually holds for a certain time period whereas an event occurs at a certain point in time with no duration.

For example, a flight may have the state departing, landing or cruising. In contrast, interesting events may be:

- A new track id is reported from the sensor.
- A track id is no longer reported.
- A plane has abnormally changed its altitude.
- Two planes have *steered* into a collision course.
- A plane has left its predefined flight way.

All these event types can be detected by AIMS and may lead to a change of state for the respective flight. Since the state of a flight has a certain duration, its determination must be robustly defined. For example, a turning plane may be on a "collision course" for some seconds, leaving this state immediately. Thus, a situation like this is more like an artifact rather than a change of status. Only if the collision course lasts for some seconds, a critical situation has occurred and a corresponding flight state should be derived. Another example is that planes may change their flight levels during their cruising period only because of noise in the underlying track data, while a landing plane may even increase for a short time in order to adjust its approach. All these events do not cause a change in the state of a flight and should be ignored.

In order to avoid this kind of false alerts, we employ the "time to live" (TTL) approach already introduced above for maintaining the delta and history table. This time, it is employed to make state derivations using our anomaly detection views more robust. For instance, let us reconsider the view for detecting landing planes from Section III. Each of the three criteria employed in the respective view – a negative vertical speed, a flight level below zero and the proximity to an airport – could be violated for a short time. Therefore, the view is materialized and a TTL value added for maintaining its content. The following set of delta rules is used to update TTL values accordingly:

$$
\begin{aligned}
\Delta^{+}\text{hist}(\text{track\_id}, \cdots) \leftarrow & \Delta^{+}\text{tracks}(\text{track\_}\textit{textid}, \cdots) \\
\Delta^{u}\text{land}(\text{id}, \text{TTL}', \text{conf}') \leftarrow & \text{landTTL}(\text{id}, \text{TTL}, \text{conf}), \\
& \neg\Delta^{+}\text{tracks}(\text{id}, \_) \\
& \underline{\text{with}}\ \text{TTL}' := \text{TTL} - 1 \\
\Delta^{u}\text{land}(\text{id}, \text{TTL}, \text{conf}') \leftarrow & \text{landTTL}(\text{id}, \text{TTL}, \text{conf}), \\
& \Delta^{+}\text{tracks}(\text{id}, \text{pos}_1, \_), \\
& \text{airports}(\text{pos}_2), v_{\text{vert}} < 3000\text{ft/min}, \\
& \text{level} < 3000\text{ft}, \\
& ||\text{pos}_1 - \text{pos}_2|| < 20\text{km} \\
& \underline{\text{with}}\ \text{TTL} := 15, \text{conf}' = \text{conf} + 1 \\
\Delta^{+}\text{land}(\text{id}, \text{TTL}, 1) \leftarrow & \Delta^{+}\text{track}(\text{id}, \_), \neg\text{land}(\text{id}, \_) \\
& \underline{\text{with}}\ \text{TTL} := 15
\end{aligned}
\tag{9}
$$

In the first block, the TTL counter is decremented. All previously existing tracks are refreshed in the second block and the confirmation counter `CONF` is increased by 1. In the following block, new data is inserted with a "fresh" time to live but with a confirmation counter of 0, meaning that the status has not been confirmed yet. Last, all "dead" landing tracks are deleted from `land`. In the GUI of AIMS, all unconfirmed landings are ignored and thus, not depicted.

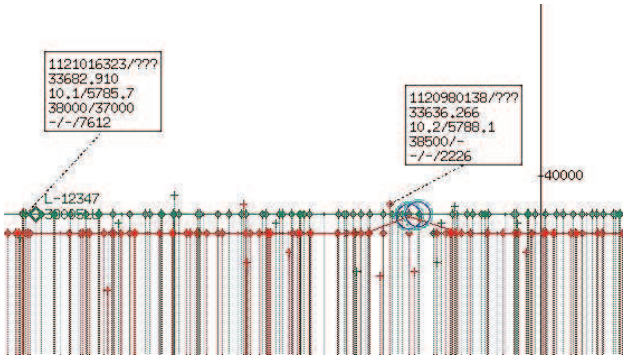

Fig. 4: Horizontal view of a critical encounter

V. Incremental Query Evaluation

The employed anomaly detection views are standard SQL views where the user defines flight anomalies in a declarative way. Internally, the continuous recomputation of these views is carried out by using update propagation methods. Update propagation has been studied mainly in the context of integrity checking and materialized views maintenance. Its application for analyzing data streams, however, has not attracted much attention so far. The key idea is to transform each SQL view already at schema design time into a so-called delta view, a specialized version of the view referring to changes in the underlying tables (data streams), only. The original view definitions are employed only once; namely for materializing their initial answers while the specialized versions are used afterwards for continuously updating the materialized results.

We adopted this idea for AIMS, using delta-view techniques for a synchronized update of detected anomalies. To this end, update statements within triggers are applied instead of the original indicator views for incrementally maintaining the materialized values. In principle, these update statements can be automatically compiled from the original views. However, we do not have a full-fledged delta view compiler yet, therefore we are performing our experiments with hand-compiled delta views for the time being.

A. Magic Updates

Because of the stream scenario, a special update propagation method called Magic Updates (MU) is applied [5]. The problem of ordinary delta views is that classical query optimization strategies - such as pushing selections - cannot deal with new selection constants dynamically introduced by the underlying data streams. For solving this problem, Magic Sets is used for enhancing the incremental view definitions. As an example consider the following algebra expression for defining the view $P(x)$ based on the relations $Q(x)$, $R(x)$, $S(x, y)$ and $T(x, z)$:

$$\begin{aligned} P(x) \leftarrow & Q(x), \neg U(x) \\ P(x) \leftarrow & R(x), \neg U(x) \\ U(x) \leftarrow & S(x,y), T(x,z) \end{aligned} \tag{10}$$

The following rule would yield the induced insertions P_i of P resulting from insertions Q_i into Q:

$$\begin{aligned} \Delta^+P(x) \leftarrow & \Delta^+Q(x), \neg R(X), \neg U(x) \\ U(x) \leftarrow & S(x,y), T(x,z) \end{aligned} \tag{11}$$

Despite of the focus on changes with respect to Q, no optimization effect is achieved with respect to the evaluation of the the negated expression $U(x)$. A possible reordering of operations by using classical rules of algebraic optimization cannot provide a better focus on the changes of Q either. However, another way is to use the small number of streaming tuples in Q_i already for determining all matching join partners by introducing two semi-joins:

$$\begin{aligned} \Delta^+P(x) \leftarrow & \Delta^+Q(x), \neg R(X), \neg U(x) \\ \text{magic\_}U(x) \leftarrow & (\Delta^+Q(x), S(x,y)), \\ & (\Delta^+Q(x), T(x,z)) \end{aligned} \tag{12}$$

Under the assumption that S and T are quite large in comparison to the size of Q_i and that there is a low selectivity of the tuples in Q_i, the argument sizes of the join and difference operator are considerably reduced. Thus, the resulting incremental expression provides a much better focus on the changes to Q.

In AIMS, MU has been employed in various anomaly detection views which are part of a multi-level view hierarchy. As an example, consider the determination of delayed flights which are involved in a critical landing:

$$\text{delayCritLand(id)} \leftarrow \text{critLanding(id)}, \text{delayed(id)}. \tag{13}$$

For determining delayed flights we have to access the usually very large table of flight plans. Additionally, costly extrapolation is necessary in order to compute the ideal position of an airplane for a given timestamp based on the sequence of waypoints in its flight plan. However, the small number of critical landings can be used as an additional selection criterion for determining relevant flight plans and waypoints, only. To this end, the determined critical landings are used to form so-called magic subqueries

$$\text{magic\_delayed(id)} \leftarrow \text{critLanding(id)}. \tag{14}$$

which are joined with the table of flight plans first:

$$\begin{aligned} \text{delayed(id, d)} \leftarrow & \text{magic\_delayed(id)}, \Delta(\text{id}, \text{id}_1, \vec{v}), \\ & \text{flightPlan(id, t}_2), \\ & d := f(t_1, t_2, \vec{v}, \ldots). \end{aligned} \tag{15}$$

Using this MU approach allowed to speed up the recomputation of anomaly detection views dramatically (see Sec. VII). The introduction of auxiliary Magic Sets containing dynamically generated selection constants can also be used for improving the focus within recursive delta views. However, Magic Sets do not always lead to an improved evaluation. Generally, its optimization effects strongly depend on relation sizes, selectivities and the chosen SIP strategy. One research goal is to develop a cost-based MUs compiler for automatically generating well-optimized delta views.

B. Aggregate Functions

As soon as stream data have been processed they become historical data which have to be stored by the system. Historical data are necessary for provenance analysis or may be relevant for continuous queries involving certain statistics. A very elegant approach for accessing historical data is the Total Recall facility of Oracle [13]. It turned out, however, that Total Recall performs rather slow such that it is only suitable for purely historical queries (e.g., count all landings on Tuesday last week).

In case a statistical query with a sliding window is to be answered, AIMS employs the incremental approach mentioned above. Generally, the new aggregate value is computed based on its former value and the new incoming stream data. However, it is sometimes necessary to keep track of the values contained in the sliding window. As an example, consider the determination of the number of flights over Frankfurt within the last 60 minutes. Due to the implicit duplicate elimination of the `COUNT` function, we have to store the contents of the sliding window in an intermediate table `tbl_OverFRA` which is maintained incrementally using the following trigger:

$$\begin{aligned} \Delta^+\text{overFRA(id, time)} \leftarrow & \Delta^+\text{tracks(id, pos, lo, la)}, \\ & \text{time} = \text{NOW}(), \\ & \text{la} < 50.80 \wedge \text{la} > 50.40 \wedge \\ & \text{lo} < 8.46.4 \wedge \text{lo} > 8.35 \\ \Delta^-\text{overFRA(id,)} \leftarrow & \text{overFRA(id, time)}, \\ & \text{time} < \text{NOW}() - 3600\,\text{s} \end{aligned} \tag{16}$$

The count update can then be performed based on the updated window content as follows:

$$\begin{aligned} & \Delta^u\text{CtOverFRA(count)} \leftarrow \text{CtOverFra(\_)}, \\ & \text{count} = \text{count} + \\ & \qquad |\Delta^+\text{overFRA(id)}| - |\Delta^-\text{overFRA(id)}| \end{aligned} \tag{17}$$

VI. Evaluation of Update Propagation Methods

For showing the feasibility of update propagation in this context, we have evaluated different propagation techniques in AIMS. In the following, we give a typical example of run-time measurements for anomaly detection views used in AIMS.

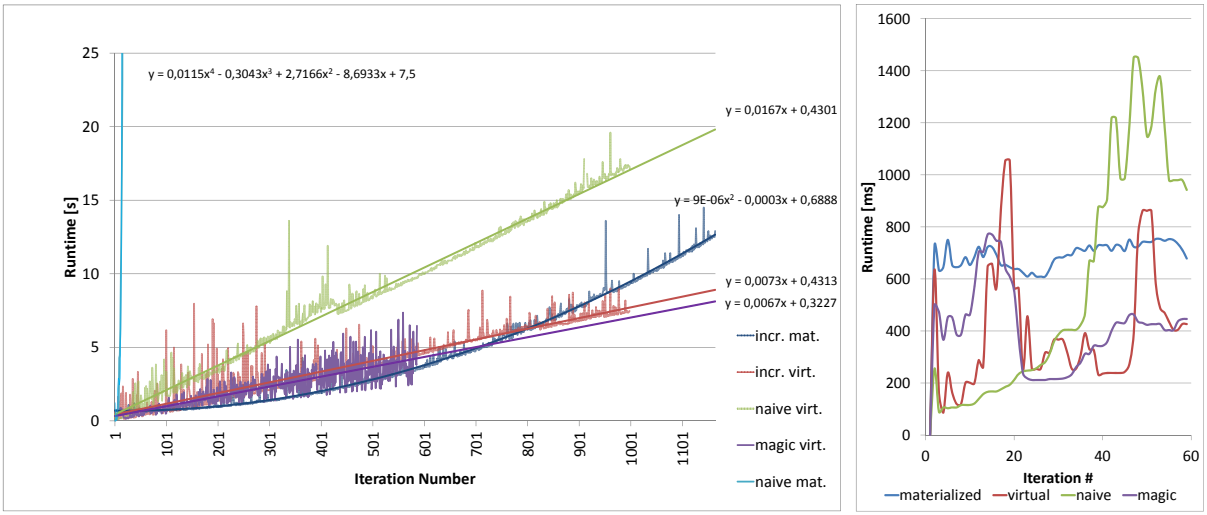

Fig. 5: Comparison of the run time of various methods. In the left image, the run times vs. the iteration number is shown. The lighter lines show the actual results, the darker lines show the trend. The formula for the trend fit is shown right hand side. The right image shows the results for the first iterations rounds (using a sliding median for evening out spikes).

A common task is the classification of flights based on the information coming from the feed of track data and the feed of flight plans.

Let us consider the stream `flights` and the query `cancelled`, where the latter is a filter on the flight plans bearing the attribute "cancelled". A query hierarchy for finding all "via"-flights then is:

$$\begin{aligned} \text{activeFlights(id)} \leftarrow & \text{tracks(id, \_)}, \\ & \neg\text{cancelledToday(id)}. \\ \text{cancelledToday(id)} \leftarrow & \text{cancelled(id, date)}, \\ & \text{date} == \text{TODAY()}. \end{aligned} \tag{18}$$

Note that the `date` attribute is needed as the flight IDs are not unique but reassigned every day. The via flights can then be selected using the statement

$$\begin{aligned} \text{viaFlights(id)} \leftarrow & \text{tracks(id, dest}_1), \\ & \text{activeFlights(id, dest}_2), \\ & \text{dest}_1, \neq \text{dest}_2. \end{aligned} \tag{19}$$

Due to the implicit self-join of `flights` in this hierarchy, a quadratic run-time could be expected. However, the common selectivity will be substantially lower than 100%. Using synthetic data (for comparability), we measured the run-time with a selectivity of 0.0004, which is equal to 400 tuples in the result set. We employed five UP methods:

- The *virtual naive* method, updating the underlying base relations and re-evaluating the complete view hierarchy
- The *materialized naive* method, storing all views as materialized, including all intermediate views.
- The *virtual incremental* method, using an incremental re-writing of the rules and feeding new data into a special delta table
- The *materialized incremental* method, using rule re-writing and materializing the adapted views.
- The *MU* method, the Magic Set transformed version of the virtual incremental variant.

A comparison of the three methods is given in Figure 5. The number of tuples inserted in every update step was fixed to 1000. In each iteration, the attribute `dateTime` was adjusted to the current date and time, so the selectivity w.r.t. the newly inserted data will be constant in each iteration and the size of the total result set will increase linearly. The measurements were done on an Intel i5-2500 @ 3.3 GHz with 8 GB RAM running Windows 7 64 bit and using Oracle 11g. For the first two methods, the query time, and for the materialized method the insert time was measured.

It can be seen that in the first iterations, the naive method will be slightly faster than the other. After about 50 iterations, the materialized incremental method will perform fast, but due to the quadratic asymptotic run-time behavior, it will perform slower after approx. 800 iterations. The virtual incremental method shows linear behavior and has half of the slope as the naive method. It has to be mentioned that the run time behavior strongly depends on the selectivity. In a scenario with a selectivity of 50%, the naive method was 1000 times slower than the virtual incremental method. The naive materialized method, however, has $O(n^4)$ runtime and is not feasible.

The asymptomatically fastest method for this scenario, the MU method, (cf. Section V-A) also shows a linear run time. Here, the offset is decreased and the slope is smaller, yielding to the best asymptotic behavior in this scenario. However, with increased selectivity, we could also obtain a larger slope for the MU than for the incremental virtual method. For instance, with a selectivity of 25%,

MU became slower than the virtual incremental method after approx. 3500 iterations.

VII. First Results and Future Work

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 [4], [14]. Therefore, various SQL extensions (e.g., [2]) and stream processing engines have been proposed (e.g. [3]) some of them even designed as full-fledged commercial products (e.g, StreamBase [14]).

As a first result, however, AIMS already indicates that incrementally evaluated SQL queries can be used for efficiently processing a realistic stream scenario. AIMS is capable of monitoring the entire German airspace by processing ≈2400 tuples (12 attributes) every 3 - 4 seconds, while most of the monitoring tasks could be solved in less than 1 second showing the feasibility of our incremental approach. It could be shown that incremental propagation and Magic Updates provide a feasible way for optimizing stream processing tasks.

AIMS could successfully identify critical situations like close encounters or deviations from the flight plan. It was interesting to notice the high number of large deviations (sometimes more than 50 miles). In addition, the high number of critical approaches was very surprising, as there were far more close encounters than expected. We could also show violations of no-fly zones and determine zones with a critical high number of aircraft movements. Currently we are working on the determination of abnormal landing approaches and even these detection views can be efficiently evaluated.

In the future, the system shall be expanded to carry out more statistics on the observed air traffic, mainly in order to improve the organization of air traffic flow and to straighten out "hot spots", i.e., regions with abnormally high air traffic. It is also planned to add a simulator component for simulating an increase in air traffic. We also want to add further anomaly detection views for discovering blind areas where radar data is typically not available. In addition, the prediction capabilities of our system ought to be employed in order to improve the dynamic models used in tracking software. For example, the knowledge about air-traffic routes can be used to predict a flight curve even if no radar data is available.

VIII. Conclusion

We have presented the airspace monitoring system AIMS which allows the detection of interesting and critical situations in air traffic using SQL views. In contrast to already existing commercial systems like FlightView [9] or FlightStats [8], AIMS provides a more flexible approach to airspace monitoring allowing the free definition of arbitrary complex events over a stream of flight data. The flexibility results from using SQL views which freely add and combine user-defined anomaly detection view specifications. In contrast, systems like OpenATC, FlightView or FlightStats do not allow for any user-defined analyzes while AirNav systems solely provides pre-defined filters, which can be freely combined though. In fact, filters in this system can be used to track aircrafts by altitude, range to a specific location or type. The system AirNav does not support, however, the detection of general emergency situations nor the identification of geographic regions with certain/critical flight statistics.

Our first performance results already indicate that conventional database systems are capable of handling this interesting geospatial stream scenario. This encourages us to believe that traditional relational database techniques are indeed suited for analyzing a wide spectrum of data streams. Another result of our prototype is the detection of a large number of critical flight approaches which underline the need for an automated monitoring tool like AIMS.

References

[1] A. Arasu et al.: *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 (2006).

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

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

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

[6] A. Behrend, R. Manthey, G. Schüller, and Monika Wieneke: *Detecting Moving Objects in Noisy Radar Data Using a Relational Database.* ADBIS 2009: 286-300

[7] T. Bosse, A. Sharpanskykh, J. Treur, H.A.P. Blom, S.H. Stroeve: *Agent-Based Modelling of Hazards in ATM* SID 2012

[8] *Flightstats*: http://www.flightstats.com/ (2011)

[9] *FlightView*: http://www.flightview.com/ (2011)

[10] *The HSQL Development Group, HyperSQL 2.2* : http://www.hsql.com/ (2011)

[11] K. Clegg, R. Alexander: *ASHiCS: Automating the Search for Hazards in Complex Systems* in SID 2011

[12] K. Clegg, R. Alexander: *Searching Air Sectors for Risk* in SID 2012

[13] *Oracle Corp.*: Oracle Database 11g Release 2 (2009)

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

[15] G. Schüller, A. Behrend, and R. Manthey: *AIMS: an SQL-based system for airspace monitoring*, in ACM SIGSPATIAL IWGS 2010, pp. 31–38.

[16] G. Schüller, R. Saul, and A. Behrend: *In-Memory Caching for Fast Stream History Access*, in ACM SIGSPATIAL IWGS 2011, pp. 37–40.

[17] G. Schüller and A. Behrend: *Towards a Universal Tracking Database*, in SSDBM 2013, pp. 10:1–10:12.