CellTrajectoryDb design overview for distributed systems seminar course

ToivoVajakas[1]

***Abstract*— The paper descibes design of high-performance trajectory data search component. Trajectory data is created by means of mobile positioning. Design goal was scalable, high-performance, high-availability architecture to be used in Demograft software solutions and in SA Archimedes grant "Real-time Location-based Big Data Algorithms".**

I. INTRODUCTION

Technical systems provide large volues of positioning data, that can be used to investigate human beharior [1]. Volume of event data is large: there can be hundreds of milliions of people generating the events and each person generates tens to hundreds of events per day. Searching the data in such large data volumes must be optimized to obtain desired performance levels.

The goal of this work was investigaton of different design options for high-performance trajectory data search component. Trajectory data is created by means of mobile positioning. Data consists of events, each event has corresponding timestamp (when event happened), pseudonymId (who generated the event), CGI (global identifier of cell that person was connected to), event type (classifier value), optionally timing advance code (characterizing distance from cell tower).

Technically one does not observe directly people but only mobile stations (mobile phones and other devices connected to radio network of mobile operatior). Other devices include AVL equipment on vehicles, security alarm devices and other remote sensors using mobile network for communication. During processing the real MSISDN of mobile station is replaced with pseudonym ID. It does not give full anonymization protection but still is useful protective measure. Later in this paper pseudonym is used to denote mobile station ID.

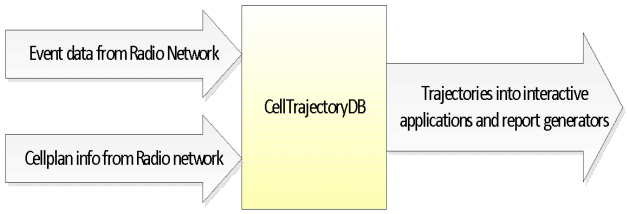

Fig. 1. Data flow into CellTrajectoryDb and out of CellTrajectoryDb.

*This work was not supported by any organization

[1]Toivo Vajakas is with Faculty of Mathematics and Computer Science, Mathematics and Computer Science, University of Tartu, Tartu 50409, Tartu tvajakas@gmail.com

II. DESIGN GOALS

A. Concrete scalability requirements

Following requirements were used as basis for design decisions

- Quick response time of single trajectory query: get quickly trajectory for given pseudonym.
- High throughput of bulk queries: for given subset of pseudonyms get quickly the trajectories.
- Statistically unbiased subsampled queries
- Covering both near-real-time recent past and historic queries.
- Design shall scale to data volumes that are present in practice
- Design shall enable easy upscaling the query throughput capabilities
- Indexed data is not affected by cellplan change
- Simple single instance setup shall be scalable enough to within one MPS Clustrer instance.
- Planned in future: optimization for high throughput full-scan queries.

B. Concrete scalability requirements

Single instance of CellTrajectoryDb is required to serve the load from single MPS Cluster instance. It shall provide following performance and scalability characteristics:

- Quick response time of single trajectory query: get quickly trajectory for given pseudonym. 50 000 days/sec from SSD (suggested configuration), 70 days/sec on HDD.
- High throughput of bulk queries: for given subset of pseudonyms get quickly the trajectories. 50 000 person-days/sec from SSD (suggested configuration), 70 person-days/sec on HDD
- Covering near-real-time recent past and historic queries. Latency from event in the stream to availability in API is 5 sec.
- Realtime event stream peak throughput at least 20 000 events/second
- Daily event volume at least 1000 000 000 events/day

III. SAMPLE PROBLEMS SOLVED USING CELLTRAJECTORYDB

A. Tracking single pseudonyms

Single person tracking is typical task in security applications used by authorities for (hopefully lawful) tracking of criminals and suspects. On has to find trajectory of one individual within a few seconds. This is easier subset of

more demanding query get trajectories of given subset of pseudonyms and the optimizations done in latter will also cover requirements of this use case.

B. Calculating trajectory-based statistics for given set of pseudonyms

Many interesting statistical values are based on trajectory of concrete pseudonyms. Sample queries:

- Find people who are in time period T1start..T1end in area A1, where are they at time period T2start..T2end ?
- For people who are in time period T1start..T1end in area A1, how often do they visit area A2 (e.g. on how many days during last 8 weeks were they visible in area A2? What is the age and gender distribution of those who visited area A2 at least on 5 days within last 8 weeks?
- Find people who are in time period T1start..T1end in area A1 and never left A1 during this time period?

This kind of questions are solved in following stages:

i defining the subset of pseudonyms (in example A: people who are at T1start..T1end in area A1)
ii analyzing the trajectories of given subset of pseudonyms (in example A: where is each concrete person from given subset at time period T2start..T2end)
iii aggregating the final result (in example A: aggregate results, eg into spatial density heatmap)

CellTrajectoryDB component is responsible for stage (ii) in this process.

C. Applying sampling for faster queries

In trajectory-based queries the stage (ii) in 3-stage process described above is most time-consuming. In case of large datasets one can often get good enough result with properly selected subset of data, thus reducing processing time. So one can use sampling select unbiased subset of all people found in stage (i), do processing of this reduced subset in stage (ii), and correct the results in stage (iii) according to sampling ratio. CellTrajectoryDB fits well for sampling, as it can quickly access individual (sampled) trajectories, query time is significantly reduced on sampled set as compared to full scan of whole dataset. Sampling introduces some uncertainty and must be applied with care. The methodology for estimating the effects of sampling is well established [2].

IV. Technical design

A. Two separate data input feeds for long-term and

Collector writes data to archives. This is most reliable data source for CellTrajectoryDb. Data upate in archives has considerable delay (up to 25 hours), additionally it requires processing time to convert archive data to properly sorted trajectory data partitions. Therefore recent history is covered by another data source. For realtime requirements there is another data feed used to fill the gap in history from latest archived data to current time. Data received via realtime feed will be ignored when the archive-based data will cover corresponding time period. Realtime feed is implemented as RMI operating over in-memory data structure. Therefore there is virtually no delay from entering the data into CellTrajectoryDb component and making it visible for queries.

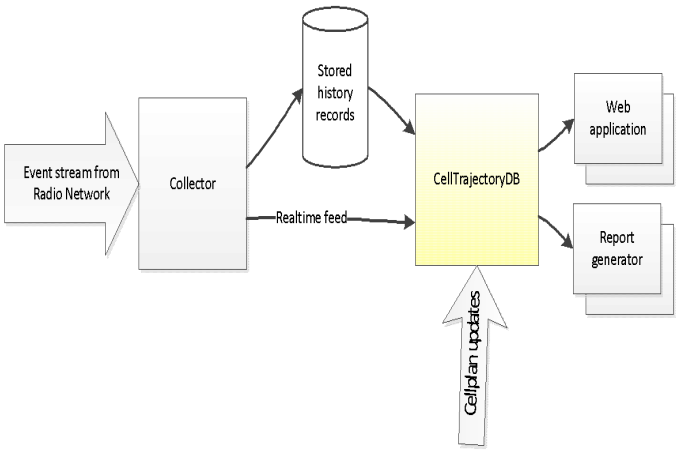

Fig. 2. Integration of CellTrajectoryDb with rest of the system.

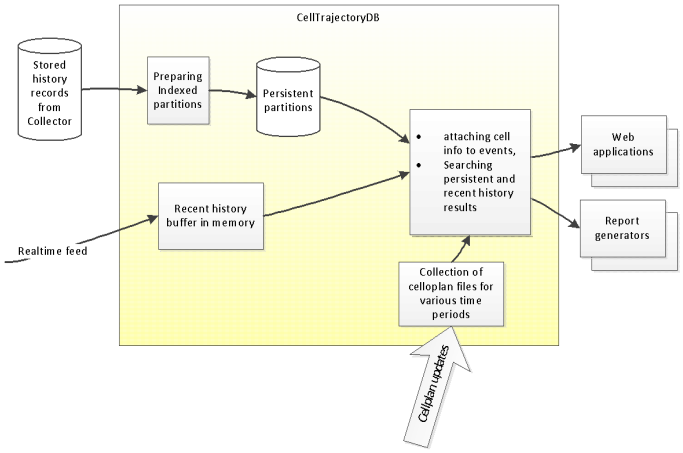

Fig. 3. Building blocks of the architecture.

B. Concurrent activities within one server instance

Within one server there are several activities going on. The server must update for

Update requests (UR) updating big chunks of data

- Add/delete partition in partitionCollection
- Add/delete cellplan in versionedCellplanCollection

UR of fine granularity

- Add new event to RealtimeTrajectoryDB

Readonly requests (ROR)

- Find trajectories
- Get metadata.

C. Synchronizing the concurrent activities within one server instance

For all activities except realtime data updates one semaphore is used, which enables all RORs in parallel. Only

one UR is enbled at once, only when no active RORs. UR is given higher priority: if UR is waiting then no new RORs are started, only currently active ones are allowed to finish.

This semaphore is not enough for all synchronization as some RORs can take considerable time and it would be not appropriate to keep real-time feed waiting this long. 2-level synchronization resolves real-time delay: all big granularity URs are done when no active ROR. Big and fine granularity URs and RORs synchronize to separate semaphore for access to event FIFO which is storing realtime events. It might be simple lock or all RORs in parallel, or 1 UR semaphore. As the FIFO lock owner will not aquire any more locks, it can always complete the critical section and there is no threat of deadlock.

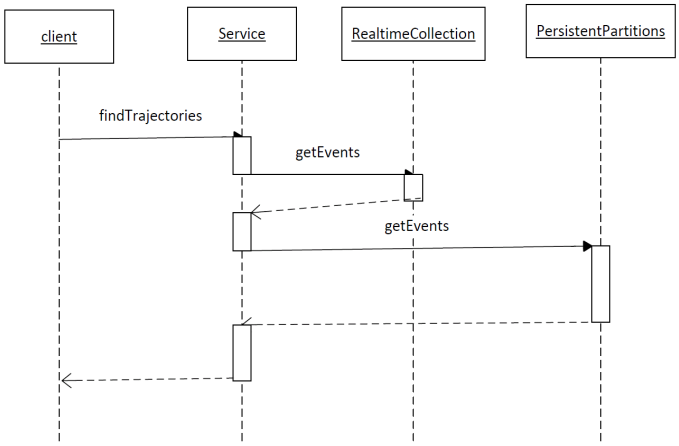

Fig. 4. findTrajectory query sequence diagram.

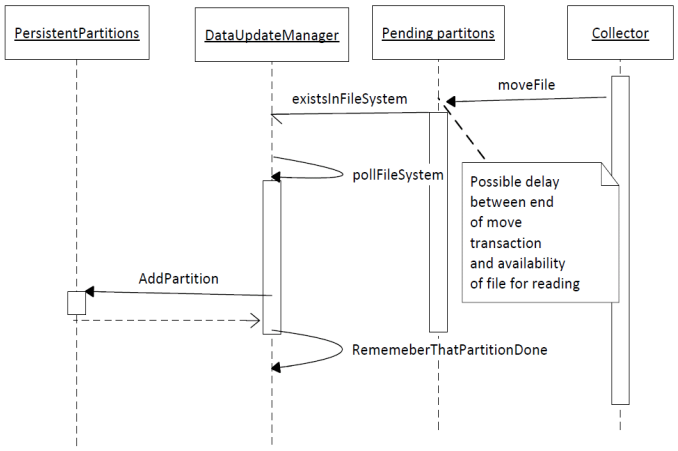

Fig. 5. New partition update sequence diagram.

V. Memory resource consumption estimates by current implementation

A. Memory usage for cellplans

Cellplan instances: each have up to 100 000 cells, each cell has geometry and other objects. If 10000 bytes per cellinfo and max 10 cellplans then 10GB for that.

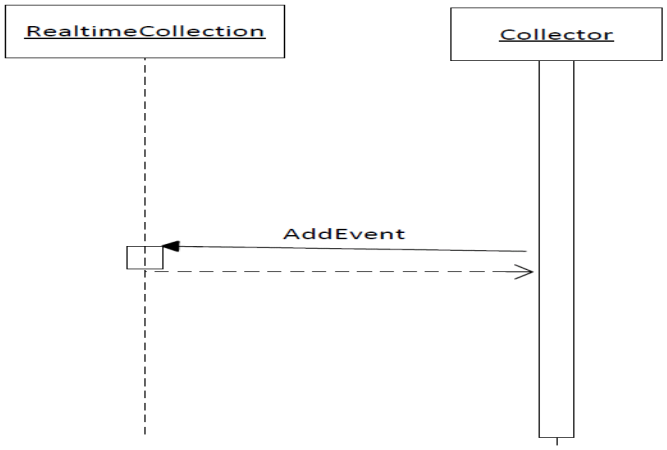

Fig. 6. Realtime events update sequence diagram.

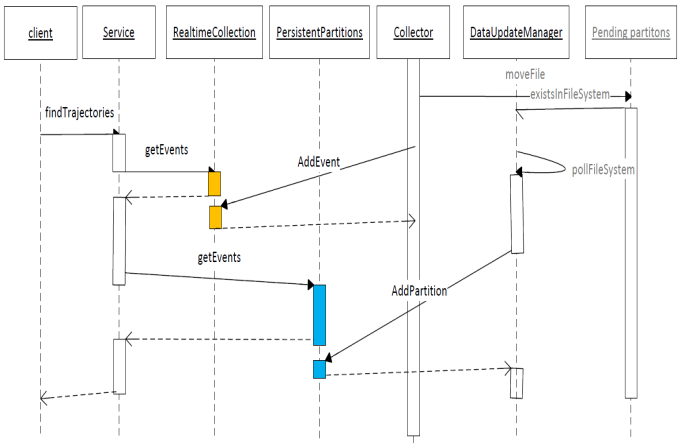

Fig. 7. findTrajectory() query sequence diagram. Race conditions solved with two different semaphores are marked with different colors

Some additional programming can reduce memory usage by order of magnitude, reusing unchanged CellInfo in different cellplan snapshots.

B. Memory usage for realtime data

- RealTimeBuffer CGI to internalId relation is implemented by trove TObjectIntHashmap. The hashmap is currently not cleaned, ie if cell CGI changes then unused old relation still uses memory. Suppose 200 000 unique CGI-s over service lifetime, 100 bytes trove TObjectIntHashmap entry. 200 000x 100=0.02GB.
- Assuming 1billion events per period that is covered by realtime buffer and 15 bytes per event, it will take 15GB for event buffer.
- Each user has 8-byte last event pointer, if we have 20 000 000 users then 0.16GB for that.

In current implementation the number of events that can be stored is limited by maximum java array size "$2^{31} - 8$".

C. Memory usage for partitioned data

- Each partition has internalId to CGI conversion (array of strings) Assuming string +pointer fits into 100 bytes

and 100 000 cells, it is up to 10MB per partition, for 33 partitions 0.33GB
- Each partition has pseudonym to seekposition index, 4+4 bytes per pseudonym. For 20 000 000 users and 33 partitions 10GB. For one year it would be 120GB.

For longer periods it is a lot of memory, mostly pseudonym to seekposition index. Some additional programming can reduce memory usage for seek index by order of magnitude without big performance overhead. Persistent partitions are already compatible with this planned improvement.

VI. Large scale solution

The design described above solves issues for one MPS Cluster site. It is planned to have separate redundant highly available instance of service at each MPS Cluster. There will be separate layer that implements query over the instances. It requires some kind of broker architecture, where separate instances will register to receive queries. This layer was discussed briefly but was not in scope of this work.

VII. Testing

The system has relatively complex behavior with concurrent for data update and trajectory queries. Therefore it needs testing against various situations. Testing fixture was programmed which makes it easy to create different situations (eg change of cellpan, adding a cellplan, deleting all cellplans visible to service, adding partitions etc. CellTrajectoryDb service component itself runs internal worker thread that is responsible for discovering changes in data partition and cellplan data in file system.

Tests were impelemented as unit tests under JUnit framework. It turned out that JUnit tests must be programmed with special care when using on-demand file creation and concurrent threads. Even daemon thread must be cleanly terminated before returning from test method, otherwise it will generate spurious errors as test resources are deleted before the [daemon] threads.

At least in Windows platform the file created in Java will not be immediately available for next file operation, eg opening the file for reading. This caused hard to detect errors and required some extra exception handling code to provide the system another opportunity to open the file if first attempt did not succeed.

The code was very unstable when test classes extended TestCase. Without this inheritance the behavior was considerably better.

Jenkins CI system also contributed to testing problems as it did sometimes mishandle svn updates (this could be corrected with repeated dummy commits).

VIII. CONCLUSIONS

The author undertook the development as it seemed to be interesting challenge to create design that provides good performance and scalability characteristics. It turned out larger work than originally estimated. I learned about some pitfalls in unit testing of components using concurrency and file creation and reading.

References

[1] M. C. Gonzalez, C. A. Hidalgo, and A.-L. Barabasi, Understanding individual human mobility patterns, Nature, vol. 453, no. 7196, pp. 779782, Jun. 2008.

[2] Thompson, Steven K., Sampling. Hoboken (N.J.) : Wiley, 2012.