# TrafficDB: HERE's High Performance Shared-Memory Data Store

Ricardo Fernandes
HERE Global B.V.
ricardo.fernandes@here.com

Piotr Zaczkowski
HERE Global B.V.
piotr.2.zaczkowski@here.com

Bernd Göttler
HERE Global B.V.
bernd.gottler@here.com

Conor Ettinoffe
HERE Global B.V.
conor.ettinoffe@here.com

Anis Moussa
HERE Global B.V.
anis.1.moussa@here.com

## ABSTRACT

HERE's traffic-aware services enable route planning and traffic visualisation on web, mobile and connected car applications. These services process thousands of requests per second and require efficient ways to access the information needed to provide a timely response to end-users. The characteristics of road traffic information and these traffic-aware services require storage solutions with specific performance features. A route planning application utilising traffic congestion information to calculate the optimal route from an origin to a destination might hit a database with millions of queries per second. However, existing storage solutions are not prepared to handle such volumes of concurrent read operations, as well as to provide the desired vertical scalability. This paper presents TrafficDB, a shared-memory data store, designed to provide high rates of read operations, enabling applications to directly access the data from memory. Our evaluation demonstrates that TrafficDB handles millions of read operations and provides near-linear scalability on multi-core machines, where additional processes can be spawned to increase the systems' throughput without a noticeable impact on the latency of querying the data store. The paper concludes with a description of how TrafficDB improved the performance of our traffic-aware services running in production.

## 1. INTRODUCTION

Traffic congestion is one of the plagues of modern life in big cities and it has an enormous impact on our society today [8, 1, 2]. Experienced by millions of commuters every day, traffic is probably the number one concern when planning a trip [28]. Smart route guidance and information systems inform drivers about the real-time traffic conditions and how they are going to impact their journey, helping them to avoid any delays caused by traffic congestion.


*Proceedings of the VLDB Endowment,* Vol. 9, No. 13


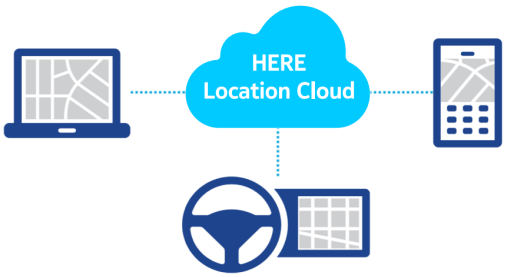


**Figure 1: HERE Location Cloud is accessible world-wide from web-based applications, mobile devices and connected cars.**

HERE is a leader in mapping and location-based services, providing fresh and highly accurate traffic information together with advanced route planning and navigation technologies that helps drivers reach their destination in the most efficient way possible.

We process billions of GPS data points from a variety of sources across the globe including smartphones, PNDs (Personal Navigation Devices), road sensors and connected cars. This data is then used to generate real-time or predictive traffic information for 58 countries and historical speed patterns for 82 countries [1]. Our road network database contains approximately 200 million navigable road segments with a total coverage of 50 million kilometres. From these, approximately 40 million road segments are covered with real-time, predictive or historical traffic, with a total coverage of 8 million kilometres. Every day, millions of users world-wide request this traffic data from web-based applications, mobile devices or vehicles connected to the HERE Location Cloud (Figure 1).

Our traffic-aware services need to be enabled with the most efficient methods to access this data. For instance, given a route planning algorithm that calculates a long continental route to a destination, it needs to be aware of how traffic incidents and congestion levels will affect the selected route. Since this calculation must be rapidly processed in order to return the result to the user immediately, the database must be able to handle high volumes of read operations with minimal access latency as route planning applications will

[1] http://company.here.com/automotive/traffic/here-traffic

hit the database with millions of queries per second. Additionally, other services such as Traffic Data Servers or Tile Rendering Servers demand for a database with geospatial features. The database must scale over multi-core architectures in order to use the available resources efficiently, and furthermore, to process thousands of requests per second.

At HERE, we face the challenge of how to methodically store this data to ensure that it is available to our applications in the most efficient manner. On the one hand we need an in-memory key-value store, able to process millions of reads per second, with geospatial features as well as optimised to scale on modern multi-core architectures, where several application processes can be spawned to increase the systems' throughput without impacting the latency of queries. On the other hand, we envision a single, common database that can be shared by all our traffic-related services. This has an enormous impact on new feature development, testing, consistency across different services, architecture simplification and ultimately costs.

After careful investigation and testing of possible solutions, we chose to design a new database from the ground up specifically optimised to solve the requirements of our traffic-enabled services. We present TrafficDB, a shared-memory key-value data store with geospatial features, optimised for traffic data storage, very high throughput with minimal memory access latency and near-linear vertical scalability. TrafficDB can also be considered as a database due to the organised structure in which the data is stored.

Today, TrafficDB is running in production at HERE in five continental regions on hundreds of nodes world-wide, with successful results in terms of performance improvements, efficient use of machine resources through better scalability, and a large reduction on infrastructure costs.

The remaining content of this paper is organised as follows: Section 2 explores related work in the field of in-memory database technology. In section 3 we describe the motivation behind the design of a new database at HERE, including the main features such a database should have in order to cover our service needs. Section 4 provides an overview on the database architecture, its functionality and design goals, as well as the details of the shared-memory implementation and its modes of operation. In section 5, we perform a set of experiments to evaluate the database performance and section 6 concludes the paper.

## 2. RELATED WORK

Although main-memory databases have been available since the early 1990s [7, 14], only over the last few years they are being conceived as primary storage instead of a caching mechanism to optimise disk based access. Current multi-core machines provide fast communication between processor cores via main memory; with the availability of large and relatively inexpensive memory units [3], it is now possible to use them to store large data sets. Relational databases were probably the first database systems to be implemented as in-memory solutions. Oracle TimesTen [17] is one of the first in-memory relational database systems with persistence and recoverability. Other systems [20, 10, 23] allow the database object to be stored in columnar format in the main memory, highly optimised to break performance barriers in analytic query workloads. VoltDB [27] is another effort to modernise and rewrite the processing of SQL based entirely on an in-memory solution. SQLite [22] is a software library that implements a self-contained, serverless, transactional SQL database engine that also supports in-memory storage.

However, today's cloud-based web and mobile applications created a new set of requirements with different levels of scalability, performance and data variability, and in some cases, traditional relational database systems are unable to meet those requirements. With the beginning of the NoSQL movement [4, 6, 18], other systems with alternative data models were developed to meet the needs of high concurrency with low latency, efficient data storage and scalability. These systems were built with the belief that complex query logics must be left to applications, which simplifies the way the database operates over data and results in predictable query performance [19].

Redis [25] is a very powerful in-memory key-value store known for its performance and is often used as a cache store. It supports various data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps and geospatial indices with radius queries. Aerospike [26], originally called Citrusleaf, is a distributed key-value store optimised for flash/SSD storage and scalability. It tries to offer the traditional database reliability, including immediate consistency and ACID, as well as the flexibility and operational efficiency of modern databases. An experimental release with geospatial store, indexing, and query capabilities is also available. Although the above mentioned databases' performance characteristics are impressive, it is not sufficient to handle the large volume of queries triggered by HERE route planning applications, where during a single route calculation, millions of queries per second may hit the database. Even these databases can take advantage of multi-threading or multiple cores, they cannot provide the necessary throughput to perform high performance route calculations. Such applications cannot afford additional latencies introduced by query languages, communication or other computational overheads. They require optimal throughput rates, which are only possible through direct memory access. Moreover, we envision a system where adding additional application processes can improve the throughput in a near-linear way without the database becoming the bottleneck.

In fact, we are now moving from generic database models to application specific models, where database systems are carefully evaluated according to the application requirements in terms of performance, scalability, transactional semantics, consistency or durability. In some cases, very specific problems demand for new technologies designed from the ground up to be optimised in order to solve the problem in question.

## 3. MOTIVATION AND REQUIREMENTS

The HERE location cloud offers a number of routing and navigation services that rely on real-time traffic data. While these services offer different products, have different modes of operation and specific requirements, all of them require access to the freshest traffic information. An application calculating a route from location $A$ to location $B$ should be aware of the traffic events impacting the route. It can use the real-time or predictive traffic conditions on the road network to calculate the optimal path. Moreover, web-based or mobile applications need to have the capability to download traffic data with minimum latency and in appropriate formats (e.g. raster images, JSON or TPEG [9]).

Figure 2 shows a central repository containing live traffic feeds that need to be accessed by the different traffic-aware services. One option is to have this repository as a central database cluster shared between all the services. When a service receives a request from an external client, it uses the network to query the database, receive the query results and respond to the client. However, a Routing Server processing a request to compute a route between two locations demands rapid access to the real-time, predictive or historical traffic conditions for a given street segment. If the application utilised the network to query the database, the execution of a routing algorithm would be extremely slow (for short routes, network connectivity latency can easily be longer than the route calculation itself). To ensure performance is not impacted, some caching mechanisms need to be put in place within the Routing Server instances to avoid queries to the database. Moreover, in order to simplify and maintain the network infrastructure, it is easier to have each Routing Server serving requests for the whole world, than to have machines serving requests based on particular scheme (e.g. geographical, where one machine serves routes in Americas, another in Australia, etc.). It then becomes crucial that traffic and map data information be easily accessible by CPUs, preferably with direct access and minimum latency, which makes main memory as the primary target. Traffic Rendering Servers and Traffic Data Servers also benefit from the performance of an in-memory database. A rendering process that already contains the traffic information in memory and can perform a spatial query to get all the locations within the bounding box of the requested tile, will process requests faster and increase the throughput. Additionally, a database storing spatial information about traffic data geometries can serve vector data that only need to be serialised on client request.

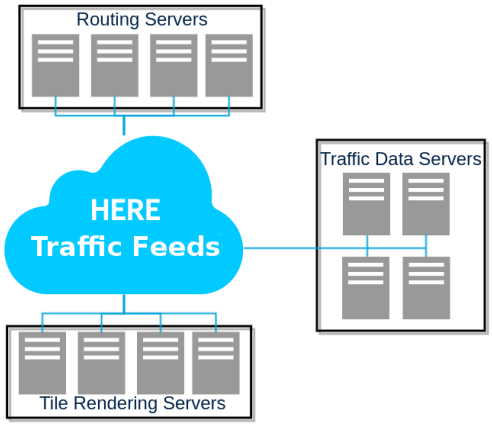


**Figure 2: Traffic-enabled services available in the HERE location cloud.**

## 3.1 Tile Rendering Service

The majority of the web mapping technologies available today are raster based. Web-based maps, such as HERE Maps (*https://maps.here.com*), consist of several map tiles usually structured in a Quadtree [24] pyramidal scheme, where the world map is split in several quad tiles[2] depending on the zoom level. Such tiles are rendered on servers and loaded by the web-browser. Creating raster imagery can be CPU and memory consuming, and in many cases the tiles can be rendered in advance, cached and then just streamed from the server. However, since traffic is a very dynamic process, constantly changing over time, real-time rendering is a strong requirement in order to provide fresh traffic information.

Our architecture consists of hundreds of Rendering Servers. Each instance spawns several *rendering processes*. Ideally, we should spawn as many *rendering processes* as CPUs available on the instance to make efficient use of the available resources. However, each *rendering processes* needs to access the traffic database. Upon a tile request, each process needs to retrieve all the traffic data within the tile; querying an external database would have detrimental effects on the *rendering processes*' performance. In order to rapidly process requests, traffic information needs to be readily available in main memory. However, storing the real-time traffic, predictive and historical traffic patterns for each *rendering process* would require an excessive amount of memory. So we would need to reduce the number of *rendering processes* in order to remain within the boundaries set by the memory capacity. For example, given an instance with 32 CPUs and 30 GB of memory, if the traffic data takes up to 5 GB, one can only spawn 6 *rendering processes*; this leaves the other 26 CPUs idle; an example of inefficient usage of available resources. Therefore, having a single in-memory database per instance that can be efficiently queried by the *rendering processes*, would allow vertical scalability and contrary to the former solution, resourceful use of memory and CPU.

## 3.2 Traffic Data Services

Traffic Data Services provide traffic information in different output formats, such as JSON, XML, TPEG [9] and other binary formats. Unlike Tile Rendering Services where the data is utilised to render an image, the data is serialised in a specified format and transferred to mobile devices or vehicles to then be loaded into their HERE-powered applications. Vector tiles containing traffic information can also be delivered though this server. The memory constraints of Tile Rendering Servers also apply here, and an in-memory database would improve the access time to traffic data and increase the throughput of the service. Spatial indexing is extremely important in order to provide fast retrieval of data for the requested area.

## 3.3 Routing Services

Route Planning Services help people to find the best path from one location to another; the 'best path' being influenced both by geometry and by dynamic conditions. In order to provide accurate route planning for a given moment in time, current traffic conditions on the road network must be taken into account. Route planning must be aware of accidents, congestion and other dynamic events such as weather conditions. Therefore, real-time, predictive or historical traffic information is embedded into routing algorithms to compute traffic-aware or traffic-optimised routes.

[2] *https://developer.here.com/rest-apis/documentation/enterprise-map-tile/topics/key-concepts.html*

When a continental route from, for example, Porto to Berlin is requested on a HERE mobile or web application, the result is displayed to the user almost immediately. Efficient route planning algorithms demand immediate access to the traffic database. During the execution of a routing algorithm, the traffic conditions of a given road segment at any point in time must be efficiently retrievable. Therefore, it is a strong requirement to have the data in memory and optimised for access by key.

Since a Routing Server instance also contains several route calculation processes, we also have the same memory constraints as in the Tile Rendering Servers. The traffic database should allow better use of the CPU resources available on the instances and provide close to linear scalability.

The Routing Servers require constant-time data access to road segments. The computation of a continental route from Portugal to Germany needs to be performed in orders of milliseconds. We need a database that can handle large volumes of lookups by road segment.

## 3.4 Data Store Requirements

Given the description of the traffic-aware applications and their main requirements, a traffic data store must be designed with the following features in mind:

- **High-Frequency Reading**. The traffic-aware applications described above perform a very high frequency of reads. The data store must be designed to provide very fast key-value access, alongside geospatial queries.
- **Low-Frequency Writing**. While traffic is a dynamic process, usually the flow conditions on a given street will not see a high frequency of changes, and for most locations, they actually remain the same for an entire day. Although a data store may contain millions of locations, their related traffic content is not regularly updated; around every minute on average. Therefore, the data store doesn't expect a high frequency of writes.
- **Low-Latency Read Operations**. Access to data when reading must be as fast as possible, leaving main memory as the only viable option. Accessing data on a SSD drive is feasible, but I/O access has higher latency and is less predictable than access to main memory. Network access is even slower and less predictable.
- **Compacted Memory Usage**. Storing real-time traffic, predictive traffic and historical traffic patterns for all road segments within a world-wide road network, preferably on one physical machine, requires a significant amount of space. To ensure that the data store remains within the memory constraints, the data must be efficiently compacted and organised.
- **Direct Data Access, No Query Language**. Using a query language that caches, wraps or converts values from data storage introduces a significant overhead for our applications. Ideally, the data store should store the data in the exact form in which it is going to be used; no mapping from raw data objects to a relational representation or another data model. Through appropriate internal data representation and operating as an in-memory data store that works with objects directly, we eliminate the overhead of duplicated data sets and the process of copying data between locations.
- **Consistent Data**. Routing algorithms require a consistent view of the traffic data during the whole route calculation, without being affected by potential ongoing updates that could put the data to inconsistent state.
- **Resilience**. In the case that any process attached to the data store was to crash, such failure should not corrupt data within the data store, prevent from serving separate requests or affect its availability.
- **Scalability**. Within a single instance, while adding additional memory or CPUs, we should be able to increase the number of applications accessing the data store without issue. As we multiply the number of applications by a factor $N$, then the number of operations the service can process should also increase by the same factor $N$. Accessing the data store should not be the bottleneck of the system, which translates to limiting use of any locks as much as possible, with the perfect case being no use of locks at all.
- **No Persistence**. Traffic content is dynamic and highly volatile. Since our traffic data is updated every minute, disk persistence is not a requirement. For example, when recovering from a crash, the persisted content would probably be outdated. Therefore, new traffic content can be downloaded from the source after recovering.
- **Geospatial Features and Indexing**. While routing servers expect fast key-value queries, tile-rendering or data servers mostly perform geospatial queries, searching all the traffic events within a specific bounding box or with a position and radius. The data store must therefore provide instant access in response to geospatial queries. It should be easy to enable or disable spatial indexing and storage of road geometries. This should result in a reduction of memory usage and time required to load and prepare traffic data.
- **Testability**. It should be easy to test as a whole and on a modular basis. Allowing all types of testing, such as functional, performance and integration to be seamless.

# 4. TRAFFICDB ARCHITECTURE

This section describes the architecture details and core functionalities of TrafficDB. Figure 3 illustrates TrafficDB embedded within a Tile Rendering Service. In this example, each available instance contains a HTTP front-end that distributes requests across a group of *rendering processes* to ensure the work is properly distributed. While processing external client requests, *rendering processes* perform queries to the TrafficDB data store, acting as *application processes*. TrafficDB is a central in-memory data store, shared by all the *application processes* in the system.

In order to process the incoming database queries from the *application processes*, a central database process could handle those requests, query the in-memory data and serialise the response back to the *application processes*. However, this approach introduces an extra overhead to access the data that is actually in memory, and additionally, as the number of *application processes* increase this request handler would

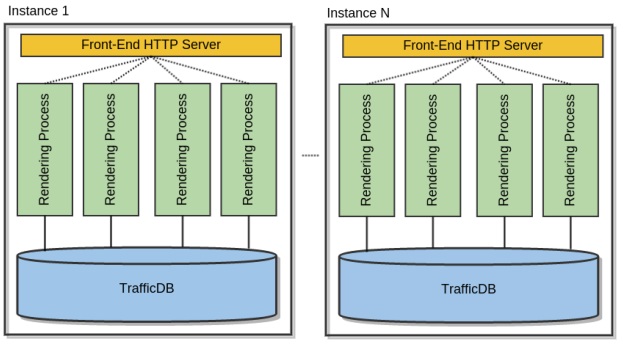


**Figure 3: Representation of Tile Rendering Server instances (virtual or physical machines), where each instance contains a TrafficDB data store shared across several application processes.**

potentially become the system's bottleneck. Therefore, no central process is utilised to translate the application queries into database-specific queries; instead, the *application processes* are directly "connected" to the shared memory data store.

## 4.1 Shared Memory Storage

TrafficDB was designed for fast read access; directly accessing the memory location of stored objects is crucial for the performance of applications, such as the Route Planning Service. Therefore, data must be stored in a region of RAM that can be shared and efficiently accessed by several different *application processes*. POSIX [13] provides a standardised API that allows processes to communicate by sharing a region of memory. Figure 4 shows the interaction between the shared memory region that stores the data and the *application processes* using it. The *daemon* is a background process responsible for managing the shared memory region, which includes creating, updating and deleting the entire data store. Being the core of TrafficDB, the *daemon* is connected to an external service that injects new traffic content. It is the only process allowed to update the data store.

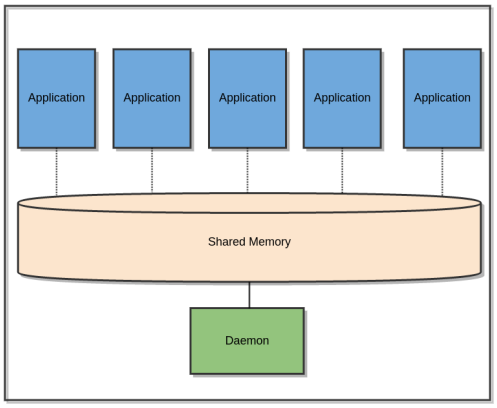


**Figure 4: Representation of the shared-memory data store updated by the daemon process and accessed by application processes.**

In the further discussion, the word "lock" is not used in the traditional sense, rather it will be used to mean two things: attaching shared memory segments into process virtual address space and increasing the kernel's internal counter of attached processes. The latter is preventing the kernel from destroying shared memory until it is closed (detached).

With a producer – consumer approach (*deamon – application process* respectively), when a consumer performs a set of queries, the data store must be locked for reading, so updates (done by producer) must wait until all the operations are performed in order to gain write access. This prevents the data from being modified whilst reading is in progress and creating possible inconsistencies, but limiting concurrent access to the data store by the *application processes* and the *daemon*. This is not to mention that possible starvation and performance degradation could occur due to lock contention, because the update process can take a few seconds and during this time no consumer cannot access the database.

To solve the above mentioned problem, TrafficDB was designed to take advantage of the double buffering scheme widely used on rendering graphics [12]. Moreover, TrafficDB utilises the Linux kernel's Shared Memory Object Management for automatic management of the objects lifetime. The *daemon* allocates a main segment in shared memory referred to as the *header*. The singleton *header* contains meta-data, such as the capacity and size of internal data structures, and any static traffic information that is known not to change (e.g. street geometry). Exluding information regarding the active *object*, only data appending inside *header* is allowed. There is also another Shared Memory Object – the *Traffic Object* (*object* for short). The *object* contains the actual traffic conditions for a given moment. It contains all the dynamic content, everything that may change periodically as the real-time traffic conditions change. Having separate shared memory objects to store the dynamic content, allows one *object* to be constantly available for the *application processes* to read and another for the *daemon* process to update. Both, *header* and *objects* are allocated to the full size upon creation of shared memory, thus eliminating memory fragmentation or a need for memory reallocation and copying.

### 4.1.1 Daemon

When the *daemon* starts for the first time the database does not exist. The *daemon* will create the *header* segment and allocate its internal data structures; loading any static data according to the database settings. If the *header* already exists, it attaches to it. Then the *daemon* enters an internal loop, waiting for traffic data updates. Whenever new traffic data is available, a new *Traffic Object* is created and the database is updated. Since only the *daemon* has access to this newly created *object*, it can write data without need for synchronisation mechanisms. The same applies to the *header*. Since new static information is appended and required only by the newly created *object*, updates can happen directly. Moreover, setting proper access rights (write for *daemon*, read-only for others ), prevents *application processes* from writing to shared memory. Additional performance enhancements could also be achieved by using shared memory with huge pages support (*SHM_HUGETLB*) enabled [21]. Once the update stage is completed the *daemon* updates the active object field in the *header* meta-data with

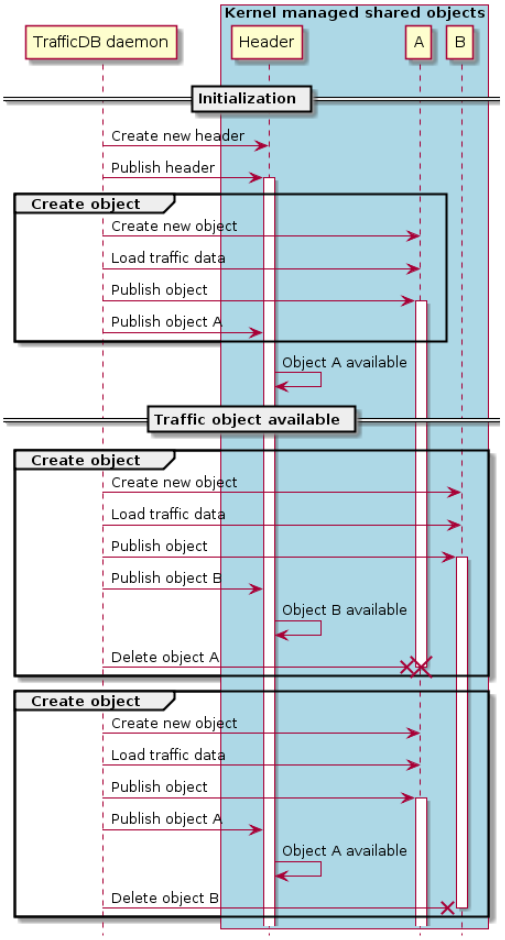


**Figure 5: Shared memory double buffering sequence**

the address of the new object. This object is now published and subsequent *application processes* connecting to the database will connect to the newly created object. The old object is marked to be destroyed. Figure 5 shows the sequence diagram of the *daemon* creating, updating and destroying *objects*. Since new traffic content is available around every minute, the *daemon* sleeps until new content is available and then the same process is executed again. The state machine diagram shown in figure 6 illustrates the *daemon's* behaviour.

### 4.1.2 Clients

Each *client* (or *application process*) retrieving traffic information from the data store needs to access the currently active *object*. Accessing consists of multiple steps that are represented by figure 7. Initially the connection to the *header* must be established in order to obtain the *id* of an active object (in case of shared memory it is a call to map it to the current process address space). After the *id* is retrieved the *client* can perform any kind of read operation on the data by simply attaching to the active object and querying the data. Given the *client* would need to execute this two step process for each operation, a negative performance impact and consistency issues would be observed. The former is caused by the fact that the attaching system call consumes

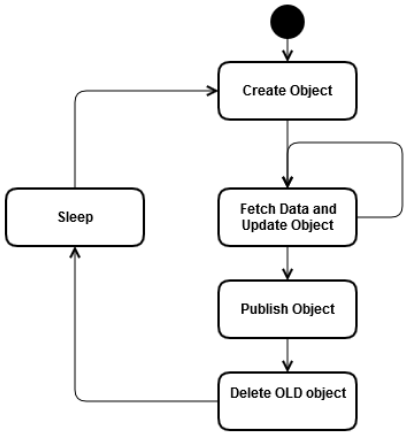


**Figure 6: State machine diagram modelling the behaviour of TrafficDB extitdaemon process.**

the majority of the computation required to complete the query. The other issue is with consistency because between two queries the active *object* can change. As a result, the same road segment may return logically inconsistent data (e.g. two different speed values). In order to overcome the previously mentioned issues the transaction concept is used, by introducing locking of *Traffic Objects*. Each client can lock an *object* and execute several read operations without the overhead of reattaching, whilst having a consistent view of the traffic data until the *object* is unlocked (Figure 8).

One should note that the Linux 'Shared Memory Object' offers a very useful feature: Once shared-memory is marked for destruction by the *daemon*, it will only be deleted by the kernel once the number of attached (connected) *clients* is zero. Thus, if there is a client still connected to this particular object (through query execution or a lock operation), the Shared Memory Object will remain in memory and exist until all *clients* have disconnected. Moreover, the same shared memory key can be reused by the *daemon* to create a new *object*, as the kernel is aware of the corresponding shared memory that was marked for destruction.

### 4.1.3 Availability and Fault Tolerance

After the database is created and published by the *daemon*, its availability is ensured by the kernel, i.e. if the *daemon* is stopped, objects remain in memory and can be accessed by *clients*. As previously mentioned a Shared Memory Object is only removed when explicitly destroyed by the *daemon* (clients are not permitted to do so). Moreover, if a client crashes, this event will not pose a threat to TrafficDB, because whenever a process crashes, the kernel will automatically close all connections and detach from all Shared Memory Objects. There is also an issue with the *clients* keeping the lock indefinitely, by executing very long operations or by simply having a bug in code, which is not handled by TrafficDB, i.e. no time-out mechanism exists. This protection is handled by the monitoring which can check the behaviour of client processes. In the case of an unexpected failure, monitoring restarts the *daemon*, which initially checks the consistency of the header and available objects. In the case that it detects any issues it will automatically destroy cor-

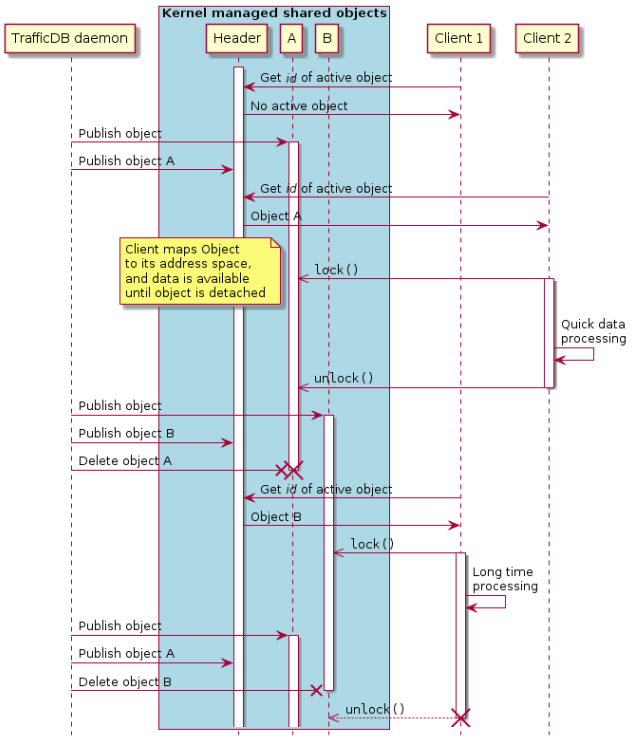


**Figure 7: TrafficDB clients to *Traffic Object* communication**

rupted objects and continue normal operation. Additional functionality can be implemented by a simple external application or the *daemon* itself to dump the database before deleting it to a permanent storage. This would create a working snapshot, which could be loaded and published to shared memory later e.g. in case of a crash and *postmortem* investigation.

#### 4.1.4 Scalability

As the amount of traffic data increases over time, we need to keep in mind that what the system can handle today, might not be the case in half a year. Although our goal was to increase vertical scalability by utilising one machine resource as much as possible, thanks to TrafficDB horizontal scalability is not an issue even in the short term. However, it is possible to distribute the data geographically, for example with instances serving data for a certain continent only. In this case the load balancer would direct requests to instances having corresponding traffic data for specific regions.

### 4.2 Shared Memory Data Structures

Shared memory allows multiple processes or multiple threads within a process to share the same data store. When the *daemon* creates a new *object* it receives a pointer to the beginning of the region where the data is going to be stored. Data structures must be carefully designed as everything must fit into the shared memory regions. When an application process attaches to the traffic object, it needs to know the starting addresses and dimensions of the internal data structures. The header contains this meta-data information like addresses, sizes and capacities. Once the process

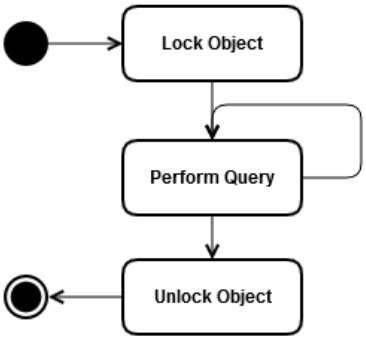


**Figure 8: State machine diagram modelling the behaviour of client applications accessing TrafficDB objects.**

attaches to the *object*, it reads this meta-data in order to assign its internal pointers to the correct addresses in shared memory region. TrafficDB provides a high level API that hides the complexity of its internal data structures, storage engine and memory management.

#### 4.2.1 Key-Value Store

Figure 9 illustrates the internal structure of a *Traffic Object* in the database. Each *object* contains a key-value store that provides fast access by key to locations and its traffic data, it also contains a spatial index data-structure, location related information such as the road segments' attributes and their geometry as well as the actual traffic data.

Since everything has to fit inside a contiguous memory region, our key-value store uses a hash table optimised to be stored efficiently within that region. First, we have a table with a number of $N$ entries and then a region that contains a set of buckets. Each entry in the table points to the first bucket within a linked-list of buckets. Besides the key and a pointer to the next bucket (if any), to make reads faster, each bucket also contains data that is common to all locations, such as their bounding box, location type and pointers to Location Information and Traffic Data sections. The performance of the hash table is strongly influenced by the hash function used to match a key to an entry on the table. Each location on a road network has a unique identifier that we use as a key to access our store. Since we know how these identifiers are generated and have full control on them, our hash function was optimised to distribute the hashed values uniformly over the table and reduce the size of chaining buckets. This allows us to not only have a hash table with an average constant-time $O(1)$ lookup performance, but also very good "worst case" performance.

#### 4.2.2 Geospatial Indexing

For spatial indexing, we use an RTree [15] data-structure also optimised to be stored in the contiguous shared-memory region. It consists of a contiguous set of nodes, where the first node is the *root*. Each node contains its minimum bounding rectangle and a set of pointers to other child nodes (if any). The leaf nodes contain pointers to the locations present in the hash table (buckets), as illustrated in figure 9. The performance of geospatial queries strongly depends on

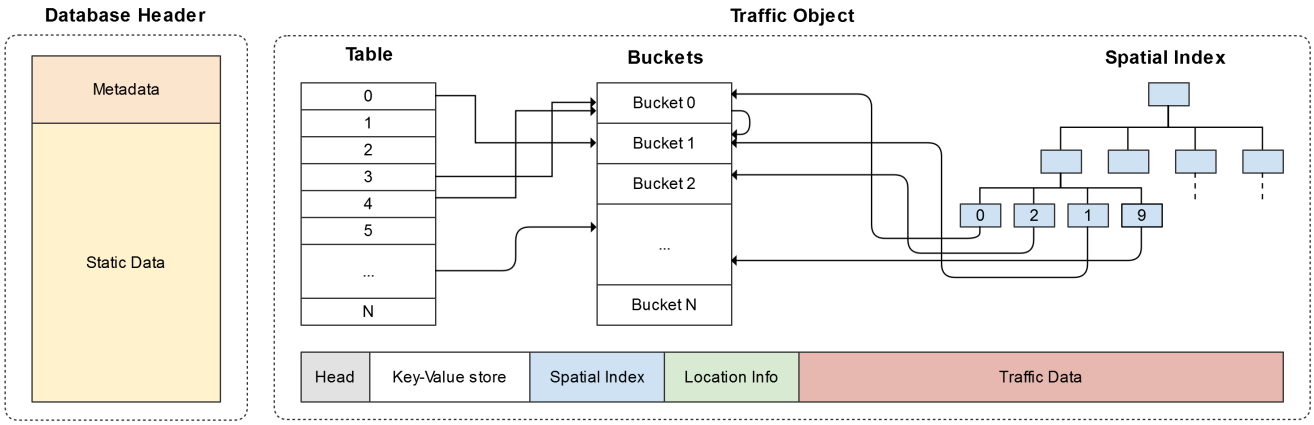


**Figure 9: The internal structure of shared memory objects. The *header* that contains the database meta-data and static data is illustrated on the left side. The *Traffic Object* and its internal data structures is illustrated on the right side.**

the balancing of the tree. Therefore, the construction of the spatial index only occurs right before the *object* is published. We traverse all the locations available in the hash table in advance, organise them and perform bulk insertion optimised to create a well balanced tree. This is performed by the *daemon* without modifying the currently active object or affecting *clients*.

### 4.2.3 *Location & Traffic Data Stores*

The Traffic Data section illustrated in figure 9 contains a blob of traffic content per location stored in the database. Each blob contains information related to the real-time congestion, incidents and predictions over the next hours for each road segment. This data is organised in such a way that enables *application processes* to perform efficient queries and check the traffic information for a specific location and at a specific time. Each location bucket within the hash table contains a pointer to the address of its traffic data blob. Blobs may have different sizes, but the data respects a specific binary schema. Due to the homogeneity of our data, we do not benefit from having a schema-less data store.

The Location Information section contains the shape coordinates and other location attributes. Although Location Data and Traffic Data have a 1-to-1 relation we separate them into two separate regions because we want the cache of locations to persist between objects. Locations can be affected by some traffic conditions on *object A*, but different traffic conditions on *object B*. Although traffic conditions change, the locations themselves do not change very often; they can remain within the cache for the subsequent objects. The entire location region is copied from the old to the new object, followed by the traffic data region being cleared from one object to another. Copying locations to the new object and resetting their traffic data pointers is an instantaneous process performed by the *daemon* without affecting other processes accessing the active object.

## 4.3 Client APIs

TrafficDB does not offer a query language, but provides a rich C++ API to directly access the data in memory instead. This API offers read-only methods to connect to the database, lock and unlock *objects*, perform spatial queries or retrieve locations by key. Listing 1 provides a code snippet of an application using the C++ API. The example demonstrates an application connecting to the database and visiting all the locations within the bounding box, serialising them to JSON format. It also retrieves the congestion level for the location with the key 12345.

**Listing 1: Code snippet using the TrafficDB C++ client API**

```
#include <trafficdb/TrafficDBClient.h>
#include "GeoJSONSerializerVisitor.h"


GeoJSONSerializerVisitor serializer;
BBox bbox( 41.10, -8.58, 41.01, -8.65 );

TrafficDBClient client;

if ( client.lock_traffic_object( ) )
{

  client.apply_spatial_visitor( bbox,
      serializer );

  const Location* loc = client.
      get_location( 12345 );
  int level = loc->get_congestion( );

  client.unlock_traffic_object( );

}
```

Besides C++, we also provide bindings for the Python language. These bindings have been useful for the integration of TrafficDB with other frameworks. We implemented several tools to collect statistics, perform real-time analysis, monitor the shared-memory data store, as well as to implement functional or performance tests. We also have implemented a REST [11] Interface Service that allows other external clients to directly query TrafficDB using a REST API. This is useful for real-time monitoring and debugging the traffic information content.

# 5. EVALUATION

Database performance is typically made up of two main metrics: *throughput* and *latency*. *Throughput* is the number of operations the database can process within a given period of time. The database *throughput* is bounded by several factors: memory and CPU, the type and size of stored data, and the operations performed on the database. One immediate sign that the database has reached its *throughput* limit is an increase on the average *latency* of queries. *Latency* is the time it takes for an operation to complete. In a distributed system there are many kinds of latencies to consider, each reflecting a perspective of the different components in play. However, since TrafficDB clients run in the same machine as TrafficDB and directly access the in-memory data-structures, *latency* is mainly affected by the presence of synchronisation mechanisms to control the access to a specific memory region. Of course, memory, CPU performance and caching, type of data and operations also impact the *latency* of TrafficDB queries.

Another important metric to consider is *vertical scalability*. As the number of applications accessing the database increases, the number of operations performed against the database also increases, and therefore we expect the throughput to increase without affecting the latency of queries.

This section provides results of experiments that are tailored to analyse the *throughput*, *latency* and *vertical scalability* of TrafficDB. The goal is to evaluate the benefits and limitations of using the shared memory architecture described in section 4, and measure the impact of lock, read and write operations on our traffic-aware applications.

## 5.1 Read Operations

In this experiment we want to evaluate the performance of the main client operations: locking, get-value queries and spatial queries. Using the TrafficDB C++ client API, we implemented a client application to measure the *throughput* and *latency* of read operations. In order to launch several clients in parallel we used OpenMP [5], a multi-platform standard library for parallel programming on shared memory systems. It simplifies the parallelism and synchronisation of clients with a relatively high level of abstraction.

To reproduce a production-like scenario, the database was updated with a real snapshot of the traffic content used in production, containing congestion information and incident events for approximately 40 million road segments worldwide and requiring approximately 5GB of RAM.

Since performance of cloud-based virtual-instances is sometimes unpredictable [16] and we want to have full control of our experimental environment to better understand the results, these experiments were performed on a multi-core machine with an Intel® Xeon® ES-1650 v2 3.5GHz processor, with 6 cores (12 threads) and 16GB of RAM. In order to achieve high confidence level in results, each experiment was performed multiple times.

### 5.1.1 Key-Value Queries

The objective of this experiment is to measure the *throughput* and *latency* of *GET* operations. The performance of these operations is strongly dependent on the performance of the internal hash table implementation and its hash function. In order to perform a fair evaluation, we need to test all the possible keys, and thus all the available locations in the database must be queried. Moreover, to reproduce a random behaviour of queries and to avoid taking advantage of CPU caching, each client should query for the locations in a random order. Therefore, as a pre-stage, we get all the keys available in the database, each client shuffles those keys in a random order, locks the *Traffic Object* and after a synchronisation point, the experiment is started.

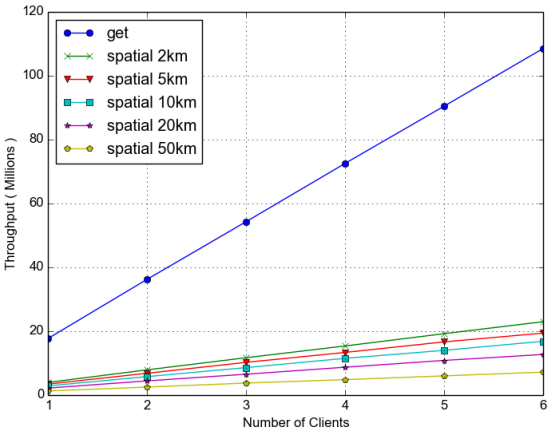


**Figure 10: Throughput of $GET$ operations and spatial queries with a radius of $2Km$, $5Km$, $10Km$, $20Km$ and $50Km$.**

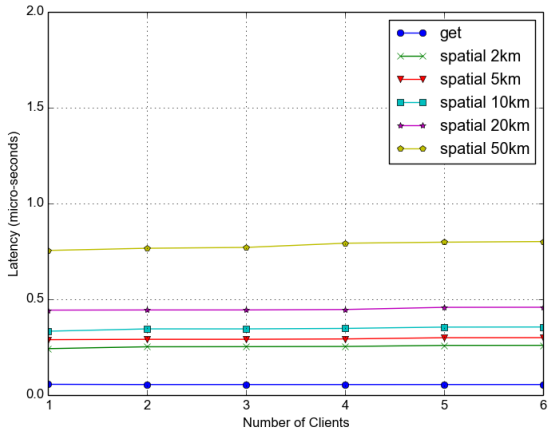


**Figure 11: Latency of $GET$ operations and spatial queries with a radius of $2Km$, $5Km$, $10Km$, $20Km$ and $50Km$.**

In Figure 10 we can observe that if we increase the number of clients accessing the database, the *throughput* of *GET* operations the database can handle also increases in a near-linear fashion. Additionally, Figure 11 shows that the latency remains constant with an increasing number of clients. This result demonstrates that *GET* operations scale very well with the shared-memory architecture of TrafficDB. This is particularly important for applications that perform high rates of *GET* operations, such as the traffic-aware Route Planning Service. Since routing processes do not affect each other while performing high rates of *GET* operations on the database, we can spawn multiple routing processes to in-

crease the system's throughput and efficiently use the available resources.

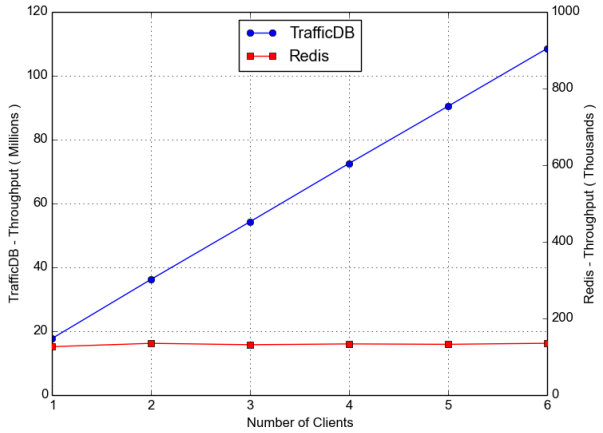


**Figure 12: TrafficDB and Redis throughput comparison of *GET* operations on a single instance deployment.**

Figure 12 shows a performance comparison between TrafficDB and Redis for *GET* operations. The purpose of this experiment is to demonstrate why existing in-memory data stores do not fulfil the performance and scalability requirements of HERE traffic-aware services. For comparison, we have chosen Redis as an example due to its reputable performance. As TrafficDB operates as a single daemon process, in order to perform a fair experiment, we also launch a single Redis process to handle all the requests coming from an increasing number of clients. Although Redis has an impressive performance, handling approximately 130 thousand operations per second, the shared memory architecture of TrafficDB brings our services to another level, making it possible to build a high performing route planning application that performs approximately 20 million operations per second. If we add another application process, it will be able to perform the same number of operations, which would double the system's throughput. While in TrafficDB we see a near-linear scaleup due to its shared-memory architecture, Redis does not allow to increase the system's throughput when adding more application processes.

### 5.1.2 Spatial Queries

A spatial query allows applications to efficiently search for locations in a specific region of the map. The performance of this operation depends on the performance of the internal R-Tree [15] index and on how the tree is efficiently balanced. Thus, in order to accurately measure the average throughput and latency of queries, we must query for locations in different regions of the map. In this experiment we split the world map into a grid of tiles, where each tile has an $N * N$ size. Then, in random order, each client performs a query for the bounding box of each tile. We tested tiles with $2km$, $5km$, $10km$, $20km$ and $50km$ values for $N$.

As we observe in Figures 10 and 11, spatial queries are naturally slower than the *GET* queries. Querying the database for large bounding boxes is also slower than querying for small bounding boxes, resulting in worse throughput for large bounding boxes. However, we can also see that the throughput increases linearly with the number of clients accessing the database, and the latency remains constant for all sizes. This validates the main goal of having *Traffic Objects* available for read-only operations. Since these operations do not require synchronisation mechanisms between concurrent clients accessing the *Traffic Object*, the system's throughput increases in a near-linear fashion with the number of clients.

### 5.1.3 Attaching/Locking Traffic Objects

As described in section 4, once a *Traffic Object* is published in the database, application clients are able to access it. The first action must be to attach to the object and lock it, guaranteeing the object is not removed until all the read operations are finished. Since this attaching operation happens at the kernel level when a process attaches to the shared memory segment, we wanted to measure its impact on accessing the database.

In this experiment our client application performed as many attaching operations as possible per second. Then we tried with several clients performing attaching operations at the same time, until a maximum of 6 clients. Figure 13 shows that attaching a *Traffic Object* is an expensive operation. When the shared memory segment is successfully attached, the system requires mutual exclusion mechanisms to update some internal variables. As we can observe, contention strongly increases latency when having more than one client performing attaching operations, affecting the overall performance of the system. Therefore, applications should perform as many read operations as possible after attaching to an object, instead of attaching every time a read operation needs to be performed.

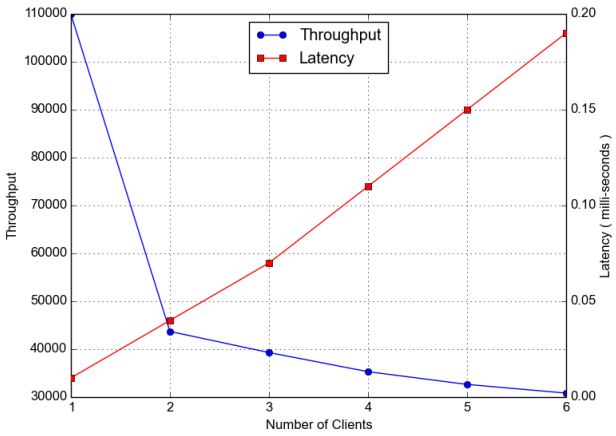


**Figure 13: Throughput and Latency of attaching operations.**

## 5.2 Scalability of Read Operations

*Scaleup* refers to the ability of processing more operations in the same amount of time but using more resources. In this experiment we want to evaluate the impact on the system if we increase the number of clients accessing the database. In equation 1, *scaleup* is calculated as follows:

$$Scaleup(P) = P * \frac{N(1) * T(1)}{N(P) * T(P)} \quad (1)$$

$N(1)$ represents the number of operations executed by a single processor and $N(P)$ the number of operations executed $P$ processors. $T(1)$ represents the execution time taken by a single processor and $T(P)$ represents the execution time taken using $P$ processors. As we multiply the number of processors $P$, the number of operations that can be executed in a given time should also increase by the same factor. Let's consider the following example: Assume it takes 1 second to perform 1 million operations using a single processor. If the number of operations increases to 3 million and $P$ also increases to 3, then the elapsed time should be the same. If the *scaleup* = 1 it means the elapsed time is the same and the system achieved a *linear scaleup*. If, for some reason it takes more time to process and thus, the *scaleup* is less than 1, the system has *sub-linear scaleup*.

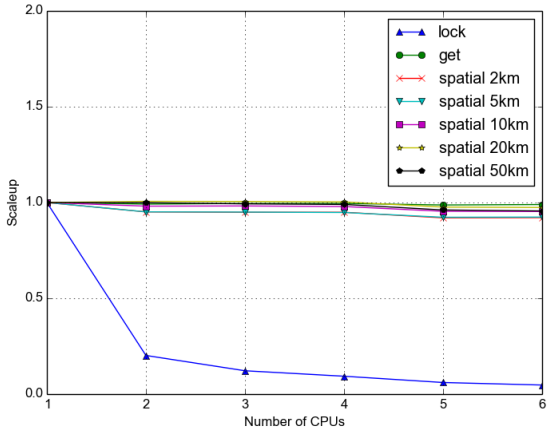


**Figure 14: Number of operations per second for spatial queries with a radius of $2Km$, $5Km$, $10Km$, $20Km$ and $50Km$.**

Figure 14 illustrates the *scaleup* of database operations for a maximum of 6 processors. We can see a near-linear scaleup for *GET* and *spatial* queries, which is explained by the advantage of not having synchronisation mechanisms. However, we see that *locking* operations do not scale and if used intensively the overall scalability of the system can be affected, strengthening the results shown in Figure 13.

## 5.3 Write Operations

As described in the previous section, write operations are performed by the TrafficDB *daemon* process only. During its life cycle, the performance of insert and update operations affects the time required to create new *Traffic Objects*. We performed some experiments using the production like traffic data to measure the throughput of these operations. Looking at Table 1, the *daemon* can perform approximately 3 million operations per second, which results in 14 seconds to publish a new *Traffic Object* containing approximately 40 million locations. However, when the locations are already present in the previous object (in cache), the *daemon* needs only approximately 5 seconds to publish a new object.

Since traffic coverage is almost the same for every *Traffic Object* created, most locations are already in the cache. However, it is also possible that some locations are affected by traffic on object $A$ but not on object $B$ and so on. In this case, locations that are not affected by traffic for more than a configurable time, are removed from the cache. We have observed a cache hit ratio of 90% in production.

| Operations | Throughput | Latency |
|---|---|---|
| Insertions | 3,067,995 | 0.32 $\mu$s |
| Updates | 8,965,017 | 0.11 $\mu$s |

**Table 1: Throughput and Latency of writes.**

## 5.4 TrafficDB in Production

Currently TrafficDB is becoming the main in-memory storage for the traffic-aware services available in the HERE location cloud. It has been successfully running in production in five regions on hundreds of nodes world-wide. The integration with Tile Rendering Services was the first successful production deployment. To provide fast response times when tiles are rendered, both world-size map and traffic data must be kept in main memory. Previously, each rendering process had a copy of the traffic content in its own memory, and spawning additional processes required large amounts of memory. In fact, we were using instances with 32 CPUs and 30GB of memory, but due to memory constraints, we could only spawn 7 or 8 rendering processes. This gave us 20 free CPUs which we were paying for but could not use. Due to the fact that TrafficDB stores traffic content in shared-memory, we are now able to run 30 rendering processes on instances with 32 CPUs in production. Since we are now using full instance capacity and each instance is now able to process 60% more requests, we could reduce the number of instances required to maintain the service live. This allowed us to strongly reduce our infrastructure costs.

Route Planning Services are also running with TrafficDB in production. In order to compute traffic-aware or traffic-optimised routes, Routing Servers now access the shared-memory database to get the traffic congestion level of road segments. Using TrafficDB, these route calculations are on average 55% faster than in the previous version, which increased the number of requests each Routing Server can process per second.

# 6. CONCLUSIONS

This paper described the main motivation behind a new database designed to solve the strong performance requirements of HERE traffic-enabled services. We presented TrafficDB, a shared memory key-value store optimised for traffic data storage, high frequency reading, with geospatial features. It was designed to scale on modern multi-core architectures, where several application processes can be spawned to improve the system's throughput. The main architecture, modes of operation and design choices were described in detail, together with a careful evaluation on the main performance metrics impacting our use cases: *throughput*, *latency* and *scalability*. Our results show that TrafficDB is able to process millions of reads per second and scales in a near-linear manner when increasing the number of clients without noticeable degradation on the latency of queries. Today, TrafficDB is running in production at HERE in five

regions on hundreds of nodes world-wide, with very successful results in terms of performance improvements, efficient use of machine resources through better scalability, and a strong reduction on infrastructure costs. Additionally, TrafficDB is used across different traffic-related services, being the core of traffic features. This move to a unified architecture brought consistency across our different services and strongly improved our development and testing phases during the implementation of new features.

## 7. ACKNOWLEDGMENTS

We would like to thank our HERE colleagues: Cristian Faundez, Rajesh Badhragiri, Jennifer Allen, Maciej Gawin, Frank Benkstein, Girish Vasireddy. Also, we would like to thank the Routing and Rendering teams for all their help in integrating TrafficDB in Route Planning and Tile Rendering services, respectively. Finally we would like to extend our special thanks to our Intellias colleagues Oleksii Zeidel and Orest Serafyn for their work on functional and performance testing of TrafficDB.

## 8. REFERENCES

[1] R. Arnott and K. Small. The economics of traffic congestion. *American scientist*, 82(5):446–455, 1994.

[2] M. Barth and K. Boriboonsomsin. Real-world carbon dioxide impacts of traffic congestion. *Transportation Research Record: Journal of the Transportation Research Board*, pages 163–171, 2008.

[3] P. A. Boncz, S. Manegold, M. L. Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[5] L. Dagum and R. Enon. Openmp: An industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[7] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. *Implementation Techniques for Main Memory Database Systems*, volume 14. ACM, 1984.

[8] A. Downs. *Stuck in Traffic: Coping with Peak-Hour Traffic Congestion*. Brookings Institution Press, 1992.

[9] B. EBU. Tpeg: Transport protocol experts group (tpeg) tpeg specifications-part 1: Introduction, numbering and versions. Technical report, TPEG-INV/002, draft (October 2002), 2002.

[10] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[11] R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[12] J. D. Foley, A. Van Dam, et al. *Fundamentals of Interactive Computer Graphics*, volume 2. Addison-Wesley Reading, MA, 1982.

[13] B. Gallmeister. *POSIX. 4 Programmers Guide: Programming for the Real World.* " O'Reilly Media, Inc.", 1995.

[14] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions On*, 4(6):509–516, 1992.

[15] A. Guttman. *R-trees: A Dynamic Index Structure for Spatial Searching*, volume 14. ACM, 1984.

[16] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168. IEEE, 2010.

[17] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[18] N. Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[19] A. Marcus. The nosql ecosystem. *The Architecture of Open Source Applications*, pages 185–205, 2011.

[20] N. Mukherjee, S. Chavan, M. Colgan, D. Das, M. Gleeson, S. Hase, A. Holloway, H. Jin, J. Kamp, K. Kulkarni, et al. Distributed architecture of oracle database in-memory. *Proceedings of the VLDB Endowment*, 8(12):1630–1641, 2015.

[21] R. Noronha and D. K. Panda. Improving scalability of openmp applications on multi-core systems using large page support. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[22] M. Owens and G. Allen. *SQLite*. Springer, 2010.

[23] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.

[24] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[25] S. Sanfilippo and P. Noordhuis. Redis, 2009.

[26] V. Srinivasan and B. Bulkowski. Citrusleaf: A real-time nosql db which preserves acid. *Proceedings of the VLDB Endowment*, 4(12), 2011.

[27] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[28] T. Vanderbilt. *Traffic*. Vintage, 2008.