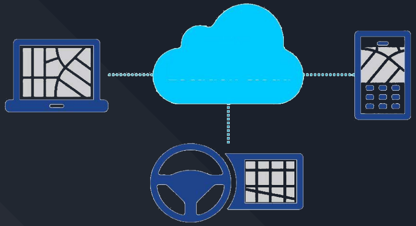
TrafficDB: HERE’s High Performance Shared-Memory Data Store

Ricardo Fernandes, Piotr Zaczkowski, Bernd Göttler, Conor Ettinoffe, and Anis Moussa

EPL646: Advanced Topics in Databases

University of Cyprus
Department of Computer Science



Christos Hadjistyllis
Marios Michael

TrafficDB - Introduction

- In-memory key-value store able to process millions of reads per second
- Supports geospatial features
- Optimised to scale on modern multi-core architectures
- A single common database shared by HERE's all traffic-related services

Introduction - HERE

- Provides accurate traffic information & advanced route planning and navigation services
- Processes billions of GPS data points across the globe
    - from smartphones, PNDs (Personal Navigation Devices), road sensors and connected cars
- Data used for generating
    - real-time and predictive traffic information for 58 countries
    - historical speed patterns for 82 countries
- DB contains 200 million navigable road segments
- Millions of users world-wide

Related Work

- Main-memory databases exist since 1990s, but only used for caching to optimise disk based access

- Oracle TimesTen, VoltDB, SQLite (supports in-memory storage), Redis and Aerospike

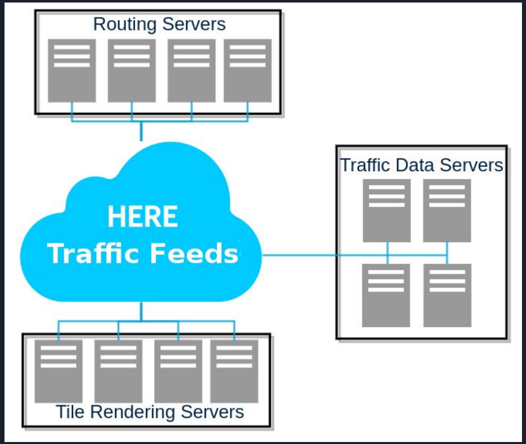
- Not sufficient to handle the large volume of queries of HERE’s route planning applications

Motivation

- Routing and navigation services rely on real-time traffic data thus require access to the freshest traffic information
- Main idea is to have a repository as a central database cluster shared between all services without latency of network I/O
- For this reason, an in-memory storage solution is the only option
    - data will be accessible by CPUs with direct access and minimum latency

Motivation - HERE's Services

- Tile Rendering Service
  - Traffic data rendered into image tiles as data changes and served to clients via web-based maps
  - CPU and memory consuming
  - Runs on Rendering Servers running many Rendering Processes
- Traffic Data Service
  - Provides traffic information in different output formats, such as JSON, XML etc.
  - Spatial indexing used for fast retrieval of data for requested area
  - Need for fast access time to traffic data and high throughput
- Routing Service
  - Give route for A to B by calculating best path
  - Must be aware of accidents, congestion and weather conditions
  - During the execution of a routing algorithm, the traffic conditions for a given road segment must be efficiently retrieved

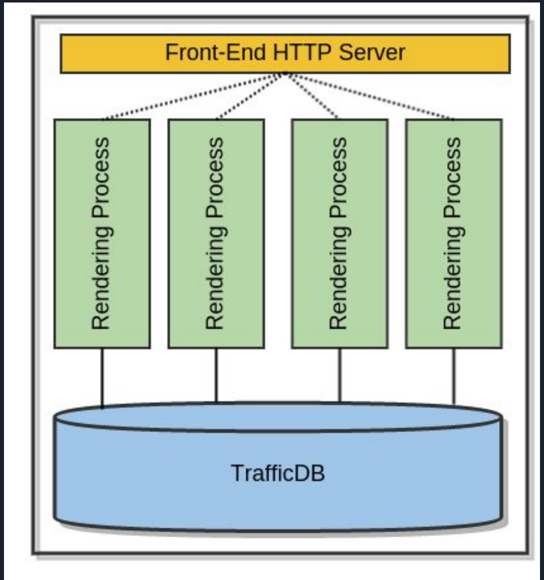


Motivation - Data Store Requirements

- High-Frequency Reading
- Low-Frequency Writing
- Low-Latency Read Operations
- Compacted Memory Usage
- Consistent Data
- Resilience
- Scalability
- Direct Data Access, No Query Language
- No persistence
- Geospatial Features and Indexing

Architecture - Tile Rendering Servers

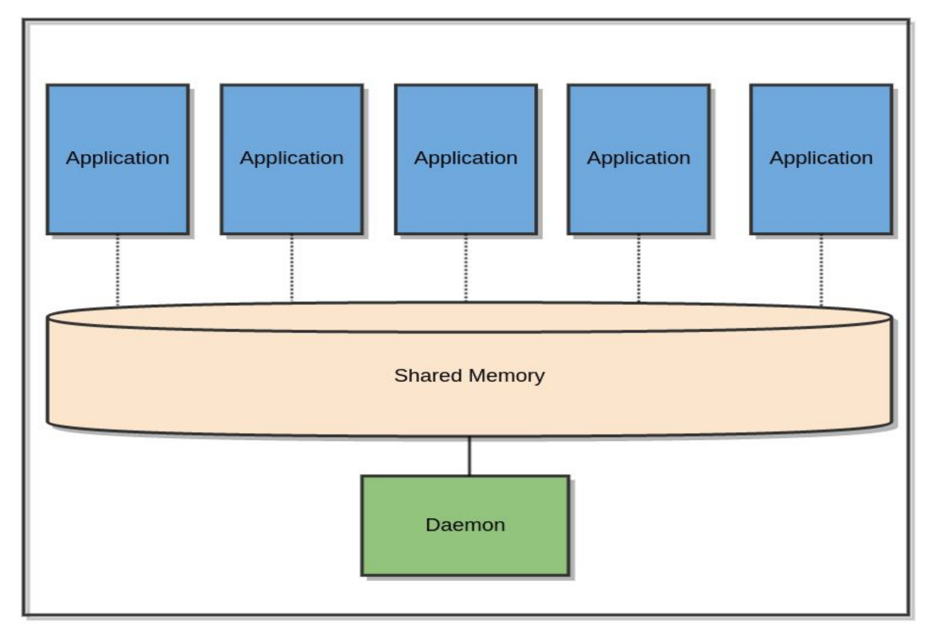
- Instances lie behind an HTTP front-end that distributes requests across a group of rendering processes
- Rendering processes are a type of TrafficDB application processes
- Application processes are directly “connected” to the data store
  - Potential bottleneck if central database process would be used for access to data store
    - Passing response messages would be a considerable overhead



Architecture - Shared Memory Storage Overview

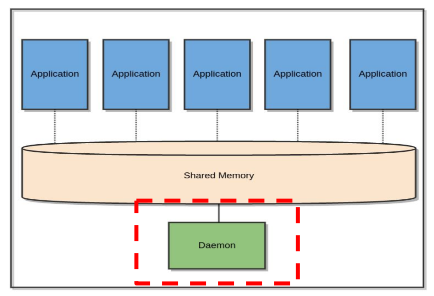
- Data are stored in a shared region of RAM that can be efficiently accessed by different types of application processes

- Uses POSIX API which allows processes to communicate by sharing a region of RAM



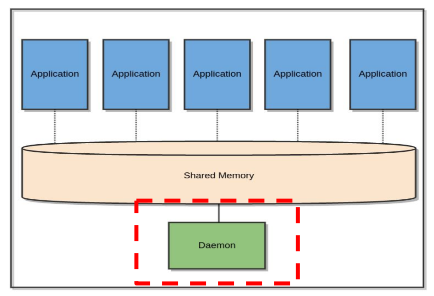
Architecture - Daemon Process (1)

- Daemon: core of TrafficDB, background process responsible for managing the shared memory region
  - creating, updating and deleting the entire data store
  - connected to an external service that injects new traffic content
  - the only process allowed to update the datastore



- Employs Producer (daemon) – Consumer (application processes) approach
  - APs lock data store for reading to perform queries
  - Daemon needs to wait to gain write access to prevent inconsistency
- But, to avoid starvation and performance degradation (lock contention) of above approach, Traffic DB uses a double buffering scheme

Architecture - Daemon Process (2)

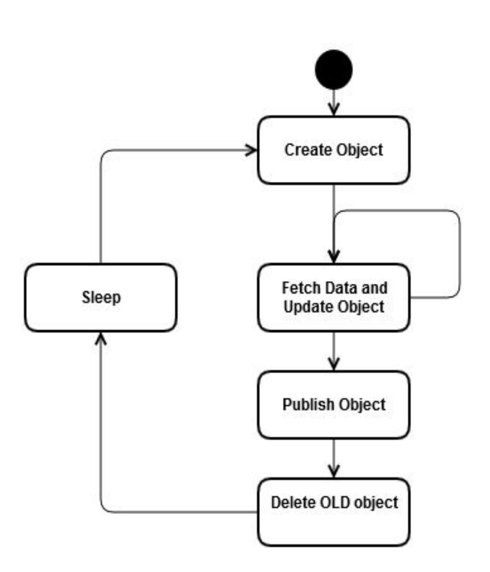


- Daemon allocates header segment
  - Holds metadata: capacity and size of data structures plus static traffic information (e.g. street geometry)

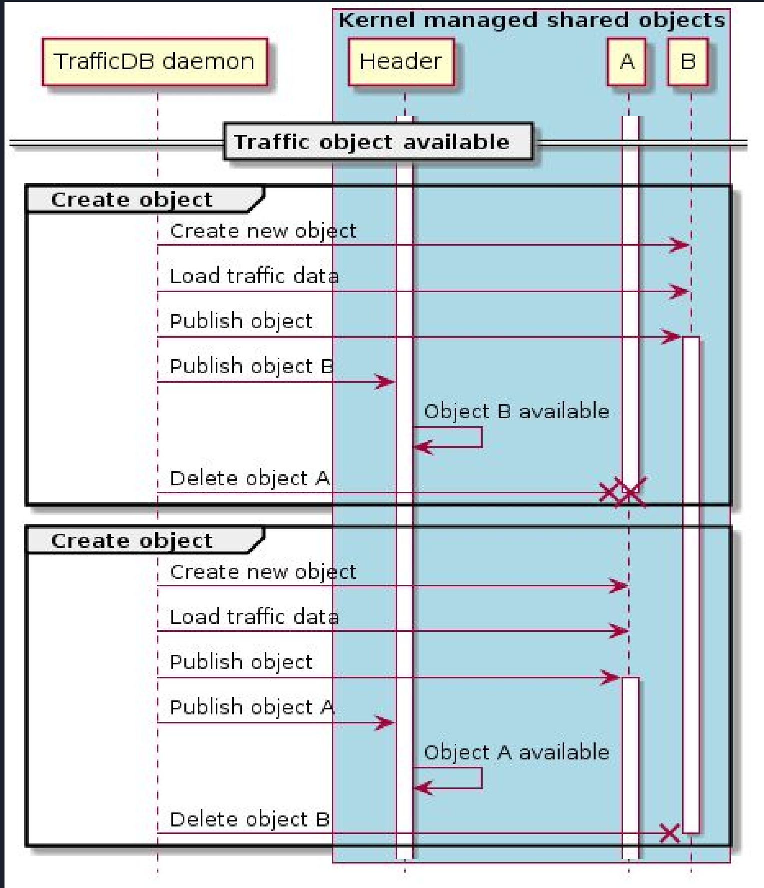
- Also creates Traffic Object: shared object containing current traffic conditions
  - Dynamic traffic content that changes along real-time traffic conditions
  - Utilizes Linux kernel's Shared Memory Object Management for automatic management of object’s lifetime

Architecture - DB Initialization + Management of Objects

- When the daemon starts for the first time the database is empty
- Daemon creates the header segment, internal data structures and loads static data from DB settings
- Then enters an internal loop, waiting for traffic data updates
- When new traffic data is available
    - A new Traffic Object is created and updated
    - Daemon updates the active object field in the header metadata with the address of the new object
    - Object is now published and APs can read it
    - The old object is marked to be destroyed



Architecture - Objects Management

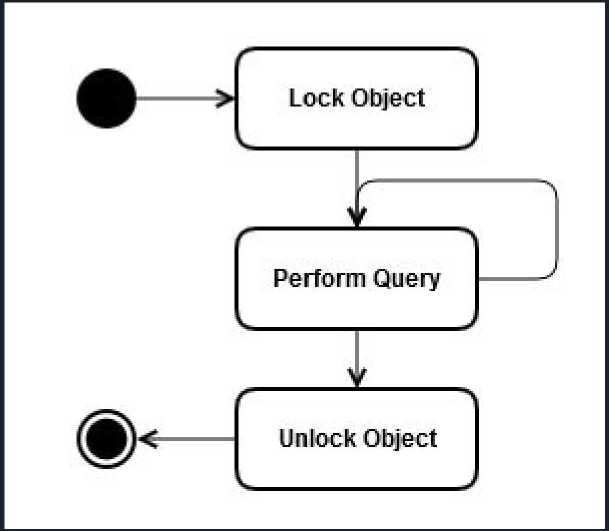


Architecture - Clients

- Each client (application process) retrieving traffic information reads header in order to obtain ID of current active object

- To avoid bad performance and consistency issues (e.g. reading old data), locking is used on the active object (field inside header)

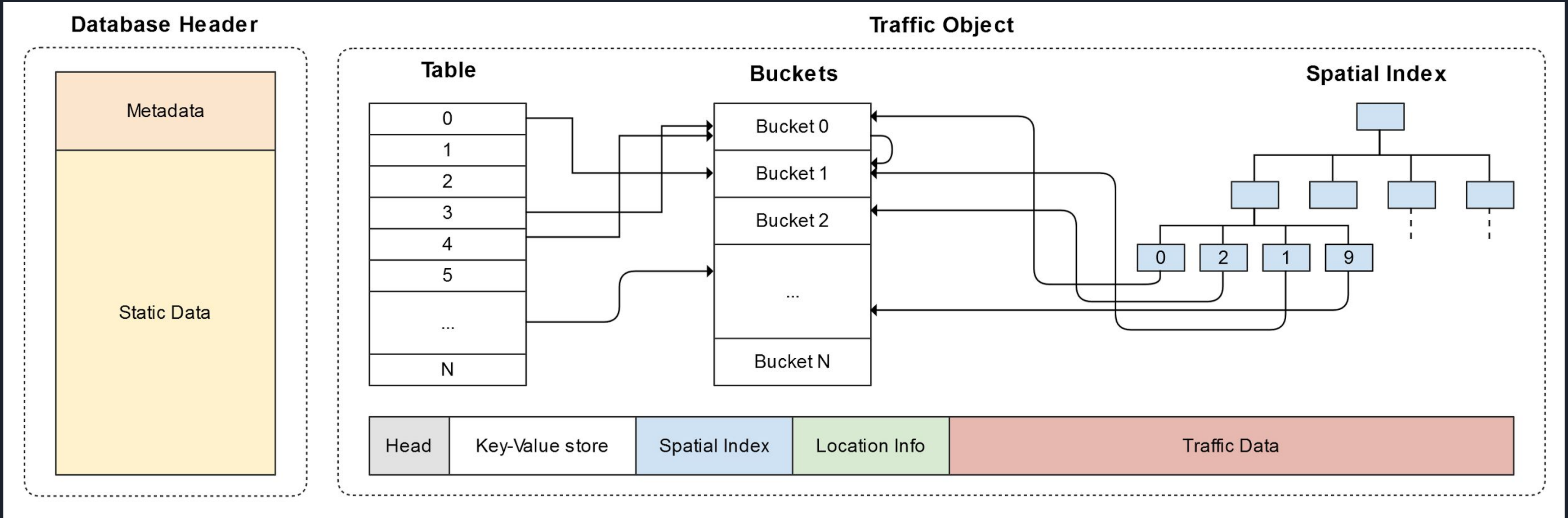
- Linux Shared Memory Object: automatically destroyed when all processes stop using it (after marked for deletion)



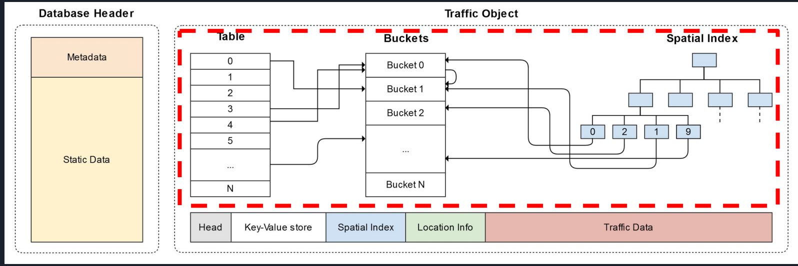
Architecture -Availability, Fault Tolerance and Scalability

- After the database is initialized by the daemon, availability is ensured by the kernel
- If the daemon is stopped, objects still remain in memory and can be accessed by clients
- If a client crashes, the kernel automatically closes all connections and detaches from all Shared Memory Objects
- Clients keeping locks indefinitely (e.g. by executing very long operations) are handled by “the monitoring” which checks the behaviour of client processes
- On unexpected failure:
  - “the monitoring” restarts the daemon
  - checks the consistency of the header and objects
  - if any issues detected, it will automatically destroy corrupted objects and continue normal operation
- Scalability: vertical (more resources) and horizontal (more machines) scalability supported

Architecture - Shared Memory Data Structures



SMDS - Key-Value Store



- Fast access to locations and traffic data
- Includes hash table where each entry points to a bucket within a linked-list:
  - Buckets include
    - pointers to respective Location Info and Traffic Data sections
    - key and pointer of next bucket
    - common data to all locations (bounding box, location type etc)
  - Key is a unique identifier given to each location in road network
  - Hash function optimised to distribute values uniformly and reduce the size of chaining buckets
- Also includes a spatial index data-structure

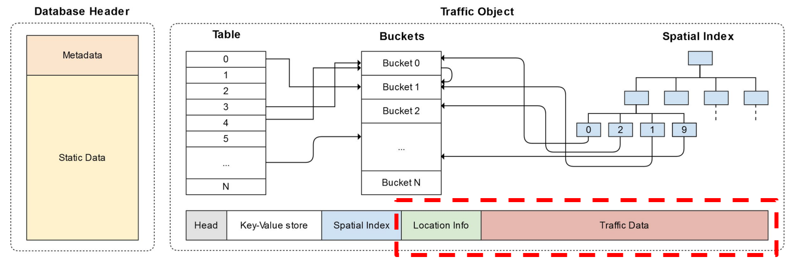
SMDS - Geospatial Index

- Uses an R-Tree data-structure optimised for storage in contiguous shared-memory regions
- Each index node contains
  - its minimum bounding rectangle
  - a set of pointers to other child nodes
- The leaf nodes contain pointers to the locations present in the buckets
- To maintain a balanced tree (for performance) index is constructed just before publishing objects
  - Daemon traverses hash table, sorts the data and does bulk insertion to create a well balanced tree

SMDS - Location Info & Traffic Data Sections

- Traffic Data section
  - Blob (binary data) of traffic content per location
  - Contains information such as real-time congestion, incidents and predictions for each road segment

- The Location Info section
  - Also a Blob, contains the shape coordinates and other location attributes



Architecture - Client APIs

- TrafficDB does not offer a query language
- Provides a rich C++ API to directly access the data in memory instead
- API offers read-only methods
  - to connect to DB
  - lock and unlock objects
  - perform spatial queries
  - retrieve locations by key
- They also provide bindings for the Python

```cpp
#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( );

}
```

Evaluation

- Three main metrics
  - **Throughput :** Number of operations the database can process within a given period of time
  - **Latency:** Is the time it takes for an operation to complete
  - **Vertical Scalability:** ratio of throughput increase while maintaining latency when scaling

- Read Operations Evaluation
  - Performance of main operations
    - locking, get-value queries and spatial queries
  - TrafficDB C++ client API used to measure performance of read operations
  - OpenMP (parallel programming library) used to launch clients in parallel
  - Database with a real snapshot of the traffic content used
    - contains information and events for 40 million road segments world-wide
    - requiring 5GB of RAM for storage
- Equipment
  - Intel Xeon ES-1650 v2 3.5GHz machine with 6 cores (12 threads) , 16 GB of RAM

Evaluation - Read Operations

- Key Value Queries
  - Measure GET operations
  - For fair evaluation, they tested all the possible keys by querying all locations in a random order to reproduce random behavior and bypass CPU caching
- Spatial Queries
  - To accurately measure the average throughput and latency of queries they queried locations in different regions of the map
  - In this experiment they split the world map into a grid of tiles, where each tile has an $N * N$ size

Evaluation - Throughput

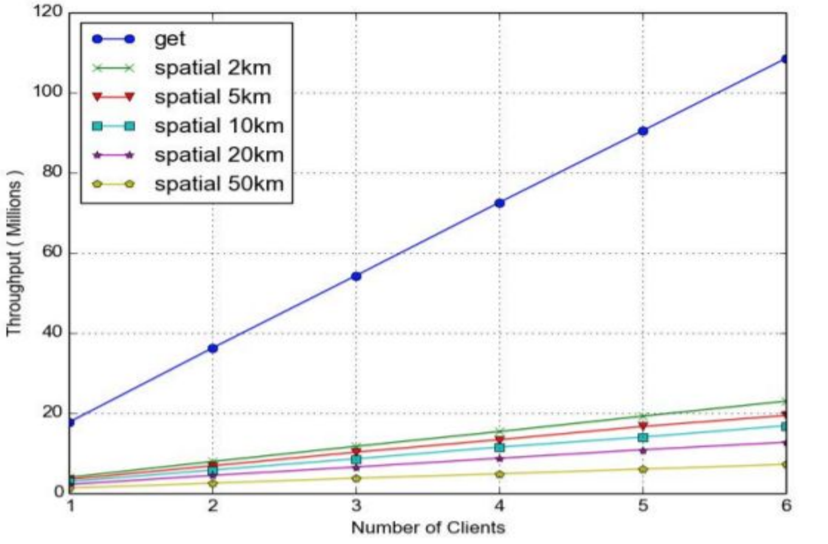


Figure 10: Throughput of GET operations and spatial queries with a radius of $2Km$, $5Km$, $10Km$, $20Km$ and $50Km$.

- GET operations: linear increase in throughput with increase of clients

- Spatial Queries: still linear but not in the same degree

Evaluation - Latency

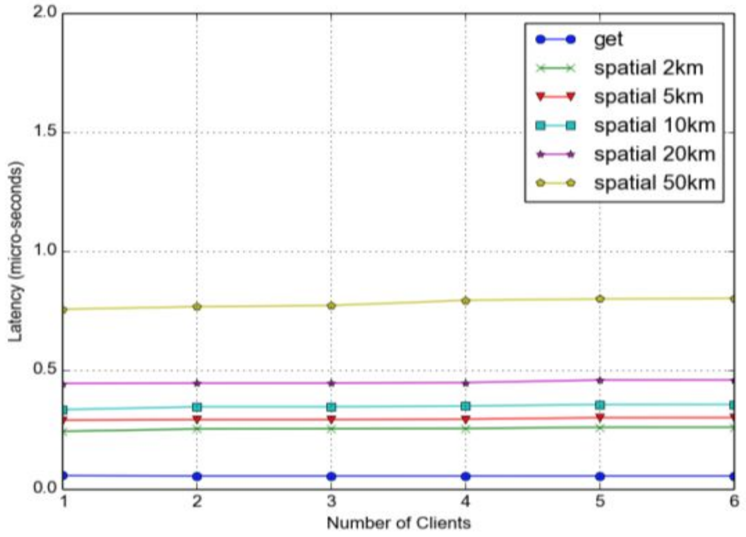


Figure 11: Latency of GET **operations and spatial queries with a radius of** $2Km$, $5Km$, $10Km$, $20Km$ **and** $50Km$.

- Latency remains relatively constant while clients increase
- This means that TrafficDB scales well under load

Evaluation - TrafficDB vs Redis

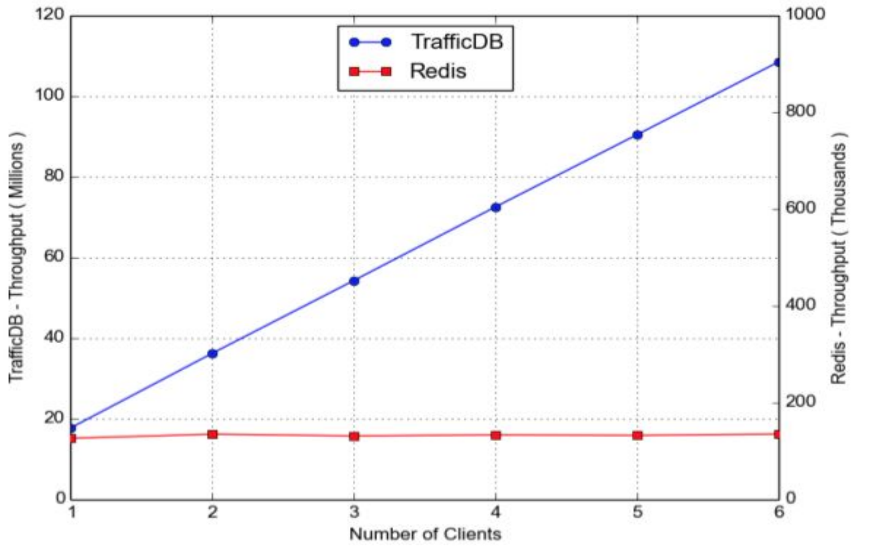


Figure 12: TrafficDB and Redis throughput comparison of GET operations on a single instance deployment.

- Redis is an in-memory data store with reputable performance

- Experiment shows that Redis (and other data stores) cannot scale like TrafficDB
    - Redis serves 130k operations per second

    - TrafficDB serves 20m operations per second

Evaluation - Attaching/Locking Traffic Objects

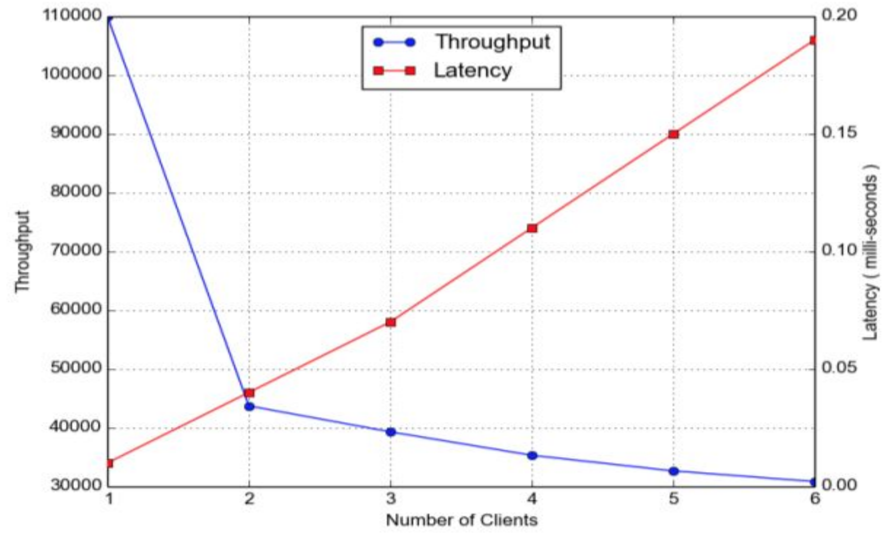


Figure 13: Throughput and Latency of attaching operations.

- As mentioned before, application processes need to lock objects when reading for consistency
- Experiment: as many locks as possible per second
- Result: latency increases due to this practise
- To minimize this, processes should perform as many reads as possible when given lock to the Traffic Object

Evaluation - Scalability of Read Operations

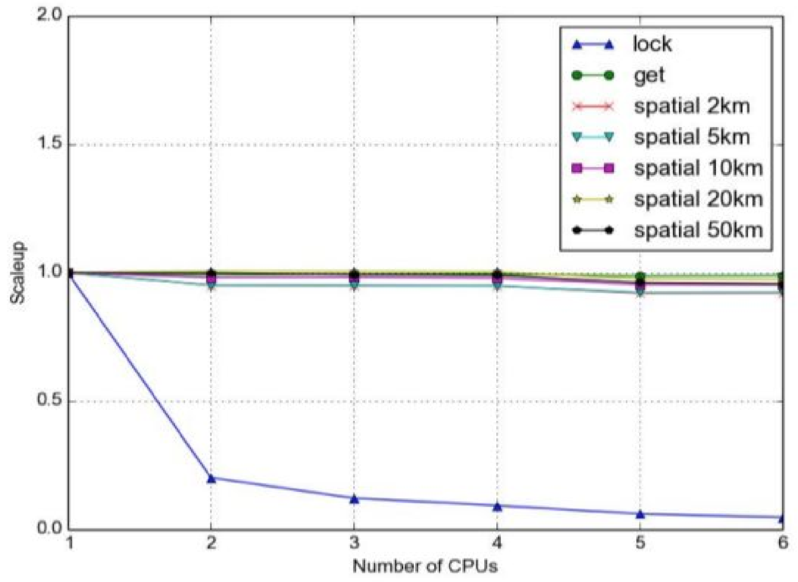


Figure 14: Number of operations per second for spatial queries with a radius of $2Km$, $5Km$, $10Km$, $20Km$ and $50Km$.

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

- N(P): Number of operations executed by P processors
- T(P): Execution time taken by using P processors
- GET and spatial operations scale well
- Locking, on the other hand proves to be a limitation for TrafficDB scalability

Evaluation - Write Operations

- As mentioned before, writes performed by Daemon process only
- The performance of insert and update operations affects the time required to create and update new Objects
- According to results below:
  - 14 seconds to publish a new Traffic Object with 40 million locations
  - 5 seconds only to update existing Traffic Object

Operations	Throughput	Latency
Insertions	3,067,995	0.32 μs
Updates	8,965,017	0.11 μs

Table 1: Throughput and Latency of writes.

TrafficDB In Production

- TrafficDB is becoming the main in-memory storage for the traffic-aware services available in the HERE location cloud

- They use TrafficDB for Tile Rendering Servers
  - Shared-memory storage allows them to run 30 rendering processes on 32 CPUs on one machine
  - Can now process 60% more requests

- They also use TrafficDB for Routing Servers
  - Route calculations are on average 55% faster than previous versions of TrafficDB

Conclusions

- Described the main motivation of designing a new database system to meet the strong performance requirements of HERE's traffic-aware services

- Introduced TrafficDB, a shared memory key-value store optimised for traffic data storage, high frequency reading, with geospatial features

- Evaluated performance in terms of throughput, latency and scalability, showing that
  - Traffic DB is able to process millions of reads per second
  - Scales in a near-linear manner on modern multi-core systems without noticeable increase in latency of read operations

References

- TrafficDB: HERE's high performance shared-memory data store Ricardo Fernandes, Piotr Zaczkowski, Bernd Göttler, Conor Ettinoffe, and Anis Moussa. 2016. Proc. VLDB Endow. 9, 13 (September 2016), 1365-1376
DOI: http://dx.doi.org/10.14778/3007263.3007274

Thank you for your attention!