CANDS: Continuous Optimal Navigation via Distributed Stream Processing

Dingyu Yang, Dongxiang Zhang, Kian-Lee Tan, Jian Cao, Frédéric Le Mouël

CANDS: Continuous Optimal Navigation via Distributed Stream Processing *

Dingyu Yang†, Dongxiang Zhang‡, Kian-Lee Tan‡, Jian Cao†, Frédéric Le Mouël§
†Department of Computer Science&Engineering, Shanghai Jiao Tong University, China
‡School of Computing, National University of Singapore, Singapore
§University of Lyon, INSA Lyon, INRIA CITI Laboratory, France
†{yangdingyu8686, cao-jian}@sjtu.edu.cn, ‡{zhangdo, tankl}@comp.nus.edu.sg, §frederic.le-mouel@insa-lyon.fr

ABSTRACT

Shortest path query over a dynamic road network is a prominent problem for the optimization of real-time traffic systems. Existing solutions rely either on a centralized index system with tremendous pre-computation overhead, or on a distributed graph processing system such as Pregel that requires much synchronization effort. However, the performance of these systems degenerates with frequent route path updates caused by continuous traffic condition change.

In this paper, we build CANDS, a distributed stream processing platform for continuous optimal shortest path queries. It provides an asynchronous solution to answering a large quantity of shortest path queries. It is able to efficiently detect affected paths and adjust their paths in the face of traffic updates. Moreover, the affected paths can be quickly updated to the optimal solutions throughout the whole navigation process. Experimental results demonstrate that the performance for answering shortest path queries by CANDS is two orders of magnitude better than that of GPS, an open-source implementation of Pregel. In addition, CANDS provides fast response to traffic updates to guarantee the optimality of answering shortest path queries.

1. INTRODUCTION

For a modern society, transportation is a major supporting infrastructure that enables the movement of people and goods. With accelerated urbanization worldwide, the number of vehicles on the road and the need for transport are growing rapidly. However, the current transportation systems, with their potential inadequacy at handling fast changing traffic conditions and the optimal control of flows of vehicles, must be rectified to accommodate increasing transportation demands. We can make use of the large amounts of GPS data to optimize the control of transportation systems. But how to effectively and efficiently process the large volume data in a timely fashion posts a substantial challenge for any modern road network system [17].

In recent years, numerous attempts have been made to address these tough problems. For example, Mobile Millennium [18] was proposed as a smart traffic estimation and prediction system. The system processes millions of real-time GPS data using cloud computing and the Spark cluster computing framework. In another case, IBM InfoSphere Streams [11] demonstrates the capability of tackling the challenges of scalability, extensibility and user interaction in the domain of Intelligent Transportation Services [17]. Both Mobile Millennium and IBM InfoSphere Streams provide traffic estimation based on real-time GPS data.

In reality, however, stringent application requirements (such as navigation) and road routing recommendations need to support various kinds of end-users including buses, taxis, ambulances, and fire-engines among others. These users should be notified as soon as possible when a traffic jam has happened on the route they will likely take. They should also be warned and recommended alternative shortest paths. Unfortunately, current systems cannot provide services such as real-time shortest paths. To this end, we aim to tackle two major challenges, by means of providing real-time shortest path services for:

(a) *Exact shortest path query processing in a dynamic graph.* Some existing shortest path approaches [28, 9] adopt a carefully designed index to achieve good performance, but the index based approach is not suited for real-time road networks due to the high cost of index update when edges change. A distributed graph processing system (e.g., Pregel [20], GPS [4] and GraphLab [16]) processes machine learning algorithm on a large graph but does not support a dynamic graph.

(b) *Route adjustment in response to traffic updates.* When an update of road status occurs, e.g., from smooth traffic to congestion, it is a demanding task to find all affected vehicles and update their shortest paths in a real-time fashion. A recent work [21] proposes two approximation techniques in the case of a traffic delay. However, it does not guarantee all the affected vehicles will be notified, nor does it provide optimal shortest paths.

We design and implement CANDS [1], which provides a continuous optimal shortest path service based on a distributed stream processing model. CANDS proposes a distributed solution with an asynchronous mechanism to answer shortest path queries in a dynamic graph. It is a hybrid scheme between index maintenance and distributed graph processing. It maintains shortcuts between each pair of border vertices in a partition. A shortcut is the shortest path between a pair of nonadjacent border vertices within the same graph partition, which is lightweight and easy to maintain in a dynamic environment.

In summary, the contributions of this paper are as follows:

*The work was done in NUS. Dongxiang Zhang and Jian Cao are contact authors.

Proceedings of the VLDB Endowment, Vol. 8, No. 2

[1]The letters come from the title Continuous Optimal Navigation via Distributed Stream Processing.

1. We propose a distributed solution with an asynchronous mechanism to answer shortest path queries in a dynamic graph. Optimization mechanisms are also applied to improve the query processing performance. For example, Message Combine method can merge relevant messages into one message to reduce network traffic; Message Broadcast strategy can quickly limit the traversal region to improve the efficiency. Furthermore, we develop an algorithm to determine the termination of our query processing and prove its correctness.

2. We propose an approach to quickly find affected queries and update their shortest path answers when the road condition changes. Two novel algorithms are designed to incrementally update the affected routes instead of re-computing the shortest path. We prove that the optimality of the shortest path can be guaranteed throughout the whole navigation process. Concurrency control is also discussed when multiple roads are updated at the same time.

3. We develop CANDS on S4 [2], a distributed stream processing system, and deploy CANDS on a cluster. Extensive experiments have been conducted to show the efficiency and robustness of CANDS. Our results show that CANDS processes shortest path queries two orders of magnitude faster than GPS [20].

The remainder of the paper is organized as follows. The problem definition is stated in Section 2. In Section 3, we present the system overview. Sections 4, 5 and 6 describe the details of *Graph Initialization*, *Query Processing* and *Query Update*, respectively. Extensive experiment results are reported in Section 7. In Section 8 existing work is reviewed, and Section 9 concludes the paper.

2. PROBLEM STATEMENT

Table 1: Notations and Symbols

G	A road network.
V	Vertices in the road network.
E	Edges in the road network.
W	The weights of the edges.
Q	The shortest path queries.
G_T	The snapshot of G at timestamp T.
W_T	The snapshot of W at timestamp T.
$\mathbb{Q}_T$	All the active navigation queries at timestamp T
$Q_{s\to t\|T}$	The shortest path query from starting location s to target location t and issued at timestamp T.
$P_{s\to t\|T}$	The result of shortest path for $Q_{s\to t\|T}$.

As a convention, we model a road network $G = (V,E,W)$ as a directed weighted graph. Each edge $e \in E$ is a road segment with a certain direction and is represented by $e = (v_i, v_j, w_e)$, where $v_i \in V$ and $v_j \in V$ are road junctions, and w_e is the average travel time to cross the edge. As real-time traffic is considered, we model G as a time-dependent graph and use $G_T = (V,E,W_T)$ to denote the snapshot of G at timestamp T.

A vehicle in a road network plays two roles: as a consumer service and as an information provider. It can send a navigation request at anytime from anywhere in the network. The request sent at time T is considered a time-dependent single-source shortest path query (**SSSP**) in G_T:

DEFINITION 1. *Time-dependent Shortest Path Query*

Given a time-dependent road network G_T and a pair of source vertex s and target vertex t, a shortest path query $Q_{s\to t|T}$ returns a path $P_{s\to t|T} = v_1 \to v_2 \to \ldots \to v_k$, where $s = v_1$, $t = v_k$, $1 \le i < k$, and $e_i = (v_i, v_{i+1}, w_{e_i}) \in G_T$, such that $\sum_{i=1}^{k-1} w_{e_i}$ is minimized.

A query $Q_{s\to t|T}$ is considered *active* from the timestamp T. When the vehicle reaches the destination t, the navigation process terminates. We mark the query *inactive* and remove it from our system.

As an information provider, a vehicle periodically sends its latest context information, including the latitude, longitude, speed and direction, to the back-end servers. There have been several existing works [27, 18] proposed to estimate traffic conditions from vehicle sensor data. In this paper, the estimation component is beyond the focus and we assume that existing solutions are adopted in our system. When a new estimation of a road segment e causes an update on its weight, we need to find which active navigation queries are affected and notify them with better shortest paths. We define this update process as a reverse shortest path query processing for edge update:

DEFINITION 2. *Reverse Shortest Path (**RSP**) Query for Edge Update*

Suppose at timestamp T, the weight of e changes from w_e to w'_e and all the active navigation queries are included in $\mathbb{Q}_T$. A reverse shortest path query finds all the $Q_{s\to t|T_1} \in \mathbb{Q}_T$ ($T_1 \le T$) which starts from s at T_1 and reaches a new location s' at T such that

1. *if $w'_e > w_e$, $e \in P_{s\to t|T_1}$ and $e \notin P_{s'\to t|T}$.*
2. *if $w'_e < w_e$, $e \notin P_{s\to t|T_1}$ and $e \in P_{s'\to t|T}$.*

The first expression identifies queries for which the affected edge, which was part of the shortest path, is no longer so at the current time. The second expression finds queries for which the affected edge, which was not part of the shortest path, is at the current time part of the shortest path.

3. SYSTEM OVERVIEW

The system overview of CANDS deployed on a cluster is shown in Figure 1. It consists of a collection of *processing elements* (PE) that are customized with specific tasks and allowed to communicate with each other via asynchronous messaging. It is worth noting that this is a basic feature supported in existing distributed stream computing platforms such as Yahoo S4 [2] and Twitter Storm [7] and our system can naturally be implemented on top of these streaming platforms.

CANDS handles two types of streaming inputs: vehicle navigation request and vehicle location update. When a navigation request arrives, CANDS needs to calculate the shortest path from the vehicle's location to its destination based on the real-time traffic conditions. We propose an asynchronous query processing strategy to reduce query latency and improve system throughput. The other type of event is the periodic GPS reading of the vehicle. These GPS data arrive at CANDS at a high rate, and are used for traffic estimation. When the average travel time of a road segment changes, existing navigation paths passing this road segment may not be optimal and need to be updated with new shortest paths accordingly.

To support optimal navigation, CANDS starts with some initialization steps. It splits the road network into smaller partitions, which are assigned to different nodes in the cluster. The weight of the edge in the road network is the average travel time and some shortcuts are maintained to facilitate query processing. These details of initialization are discussed in Section 4. The functionalities of the PE are also illustrated in Figure 1. It consists of two major components: 1) a query processing engine to handle the time-dependent SSSP queries (details in Section 5) and 2) an incremental update engine responsible for traffic status estimation and sending notifications to affected vehicles(described in Section 6).

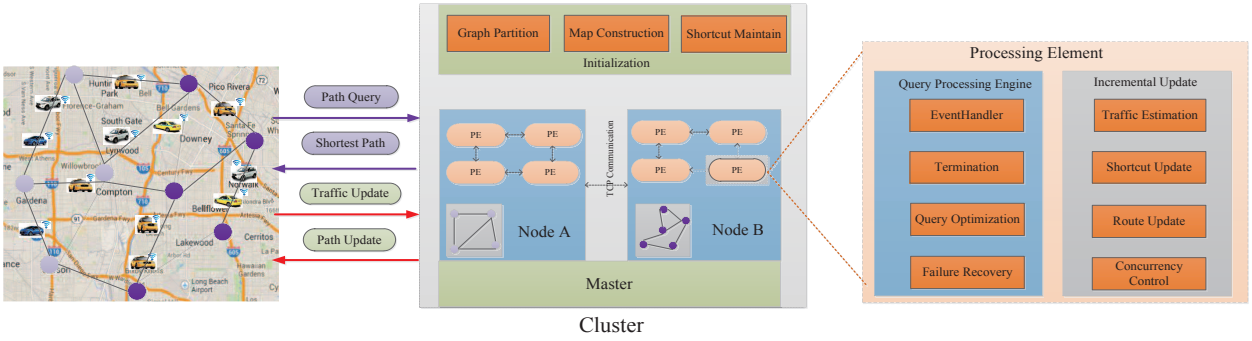


Figure 1: The system overview of CANDS

4. GRAPH INITIALIZATION

In CANDS, the road network is modeled as a time-dependent graph. The edges are road segments and vertices are road junctions. The weight of each graph edge is estimated as the average travel time to cross the segment. In the following, we present some graph initialization steps before query processing.

4.1 Graph Partitioning

In graph partitioning, the whole road network is first partitioned into M sub-graphs using a METIS-balanced graph partitioning algorithm [19] [2], where M is the number of machines in a cluster. This step is critical to performance improvement. First, the query processing time is normally determined by the slowest task and a balanced partitioning can eliminate the performance bottleneck. Second, communication between vertices in the same sub-graph is done in the same machine and network I/O can be significantly reduced. In the second level, each sub-graph is further split into N smaller partitions and each partition is assigned to one *processing element* so that the computing resources in each machine can be fully utilized. The number N is a user-defined parameter and will be estimated in Section 7. The border edges crossing two partitions are stored in both partitions because our query processing algorithm requires communication among neighboring partitions.

EXAMPLE 1. *Figure 2 shows an example of a graph split into three partitions and assigned to different PEs. For one partition G_i, we have a list of pairs $L_i = \{\langle v_i, v_j, G_i\rangle\}$ to store the border vertex b from G_i to its neighboring partitions. For example, the border vertices in partition G_1 are $\{v_1, v_4\}$ and we have $L_1 = \{\langle v_1, v_6, G_2\rangle, \langle v_1, v_5, G_2\rangle, \langle v_4, v_8, G_3\rangle, \langle v_4, v_5, G_2\rangle\}$.*

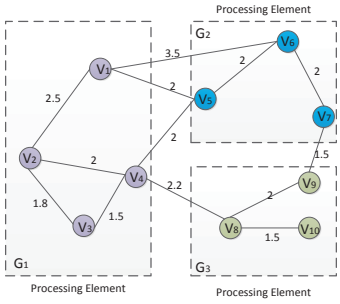


Figure 2: An example graph with three partitions

[2]Other edge-balanced graph partitioning methods can also be applied

4.2 Graph Shortcut

We maintain a collection of shortcuts to ease query processing. A shortcut is the shortest path of two border vertices within each partition. For any two border vertices b_1 and b_2, we pre-compute their shortest paths using Dijkstra algorithm and store the results. In the above example, the shortest path $v_1 \rightarrow v_2 \rightarrow v_4$ with distance 4.5 is maintained in G_1. Note that this path only guarantees local optimality instead of global one. A better result may be found to contain vertices from other partitions. In this example, $v_1 \rightarrow v_5 \rightarrow v_4$ with distance 4 is the real shortest path from v_1 to v_4. When an update occurs in the road status in this partition, the shortcuts will be refreshed to ensure the correctness of local optimality.

5. TIME-DEPENDENT SSSP QUERY PROCESSING

Existing solutions on shortest path query processing form two extremes in terms of the index maintenance cost. The index based approaches [10, 12, 28] are very efficient. However, they are difficult to adapt in a dynamic graph due to their prohibitive precomputation costs. On the other hand, shortest path algorithm is adopted in several distributed graph processing systems [20, 16, 4], and it does not have any index construction cost. But these systems cannot efficiently support shortest path queries in a real-time manner for a huge road network.

The solution proposed in this paper is a hybrid schema between these two extremes. The road network is split into partitions that are assigned to different nodes. In each node, a number of PEs are deployed, each in charge of one partition. In each partition, we maintain some shortcuts between border vertices to facilitate query processing. The shortcuts can be used directly to pass through one partition instead of traversing all the vertices in this partition.

The query processing engine shown in Figure 1 has the following functionalities. *EventHandler* is responsible for receiving events, emitting events to downstream PE. A termination algorithm (*Termination*) is designed to determine whether the resultant shortest path is optimal. In each PE, query processing optimization algorithms (*Optimization*) are further applied to improve the performance.

5.1 Query Processing Algorithm

The query processing strategy, depicted in Figure 3, relies on the coordination of four types of PEs. *QueryPE* accepts all the navigation requests from vehicles. It stores the graph partition information and knows which partition contains the source vertex of the query. A new *RouteEvent* is created and sent to *ShortPathPE*. The event contains eight fields, as listed in Table 2. The *qid*, s and t are derived from the navigation request. The border node b is initialized to be the same as s and the neighboring partition is initialized

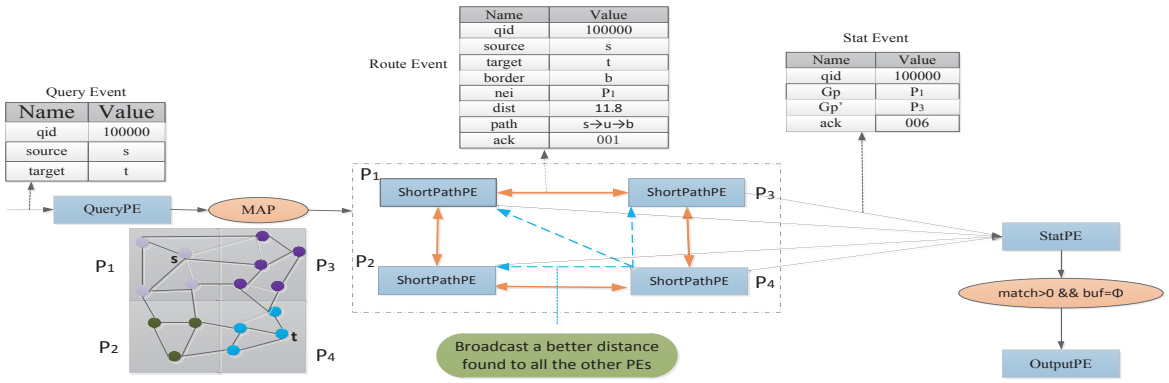


Figure 3: PEs handle a shortest path query

as the one containing the source vertex. The remaining fields are left empty. When the *ShortPathPE* receives the message, it starts cooperating with other *ShortPathPE*s (graph partitions) in an asynchronous manner to find the shortest path. Meanwhile, acknowledgements are sent from *ShortPathPE* to a *StatPE*. The *StatPE* collects the information, determines the termination of the query and sends the shortest path to *OutputPE* after the termination. The *OutputPE* then returns the result to the querying vehicle.

Table 2: The fields contained in a *RouteEvent*

qid	a query id to uniquely identify the query.
s	the source vertex of the query.
t	the target vertex of the query.
b	the border node receiving the message.
nei	the neighboring partition which sends the message.
dist	the partial best distance from s to b.
path	the shortest path from s to b.
ack	an acknowledgement sequence from the neighboring partition.

ShortPathPE is the core PE. It cooperates with neighboring partitions to find the shortest path. The idea is similar to Dijkstra algorithm, but without synchronization in each iteration to check which is the best vertex to visit. Our algorithm starts from the graph partition containing the source vertex, denoted by G_s. The *ShortPathPE* spreads the shortest path from s to all the border nodes in G_s to its neighboring partitions. The neighbors receive the events, improve the partial results and further disseminate them to neighbors. Finally, all these partial results will arrive at the graph partition containing the target vertex, denoted by G_t. When there is no message propagating in the network, our algorithm can terminate with the correct shortest path, which will be proved in the appendix.

The pseudo-code of event processing in a *ShortPathPE* with regard to a graph partition G_p is illustrated in Algorithm 1. In the following, we explicitly explain how a *ShortPathPE* handles an arrival *RouteEvent*. When an event is received from a neighboring partition, the *ShortPathPE* needs to check whether the source vertex s or the target vertex t in this event is contained in G_p, which leads to four cases:

- **Case 1:** $s \in G_p \wedge t \in G_p$ (lines 4-12). If both nodes are within G_p, we can directly calculate the shortest path from s to t in the subgraph G_p. The resultant path is sent to *StatPE*. However, the algorithm cannot be terminated at this point because the true shortest path may be missed. For example, as shown in Figure 2, the shortest path from v_1 to v_4 is $v_1 \rightarrow v_5 \rightarrow v_4$ while the Dijkstra algorithm in partition G_1 returns $v_1 \rightarrow v_2 \rightarrow v_4$. Therefore, we still need to send the query and partial result to the neighboring partitions even though we have obtained an initial candidate path.

- **Case 2:** $s \in G_p \wedge t \notin G_p$ (lines 13-19). If only s is in G_p, we call Dijkstra algorithm to calculate the shortest path from s to all the border nodes. Then, all these partial results are sent to neighboring partitions of G_p.

- **Case 3:** $s \notin G_p \wedge t \notin G_p$ (lines 20-28). If the current partition is just a bridge between s and t, the message is updated by taking account of the shortcut between border nodes and then forwarded to the neighbors. Such a forwarding process is efficient as the shortest path between border nodes has been pre-computed and is available in the local memory. Meanwhile, the partial distance from s to the incoming border node, denoted by $\delta_{s\rightarrow b}(G_p)$, is cached for future pruning. Another advantage is that it can be used to avoid message looping between two neighboring partitions, which generates partial results with loops.

- **Case 4:** $s \notin G_p \wedge t \in G_p$ (lines 29-39). If messages containing partial results arrive at the target graph partition, we call Dijkstra algorithm to calculate the path and notify *StatPE* of the result.

5.2 Query Processing Optimization

Algorithm 1 not only introduces four cases of processing an arrival *ShortPathPE*, but also incorporates two optimization strategies *Message Combiner* and *Message Broadcast* to improve the processing performance.

5.2.1 Message Combiner

After traversing one partition, messages are sent to neighbor partitions through border edges. For each border edge, one message will be generated and sent to the connecting neighbor partition. It is possible that a partition has multiple border vertices connecting to the same neighborhood. For example, in Figure 2 there are two border vertices v_1 and v_4 in partition $G1$. v_1 has to send two messages to partition G_2 and v_4 sends one message to G_2 and one message to G_3. This message dissemination mechanism incurs too much communication cost in the network. To tackle this problem, we propose a *Message Combiner* technique to combine some messages together. We add a message combiner before emitting the messages to the neighboring *ShortPathPE*. If the messages in one partition are sent to the same neighbor partition, they will be merged together as one message (lines 19 and 28 in Algorithm 1).

Algorithm 1 *RouteEvent* handling algorithm in *ShortPathPE*

1. $s \leftarrow event.source$; $t \leftarrow event.target$; $b \leftarrow event.border$;
2. $dist(s,b) \leftarrow event$.dist;$ack = nextAck(event.qid)$;
3. initialize a *StatEvents sEvent*; $sEvent$.rev$[G_p] \leftarrow 1$;
4. **if** $s \in G_p$ && $t \in G_p$ **then**
5. dist(s,t) $\leftarrow$ Dijkstra(s,t)
6. **if** dist(s,t)$< \delta_{s\to t}$ **then**
7. initialize a *PathEvent pEvent*
8. $pEvent$.path $\leftarrow path(s,t)$
9. $\delta_{s\to t} \leftarrow dist(s,t)$
10. broadcast $\delta_{s\to t}$ to all the other *ShortPathPE*
11. emit *pEvent* to *StatPE*
12. $T \leftarrow T \bigcup \langle qid, G_p, G_p, ack\rangle$
13. **else if** $s \in G_p$ && $t \notin G_p$ **then**
14. **for** each $b' \in L_n.keySet()$ **do**
15. run Dijkstra(s,b')
16. initialize a *rEvent*
17. $rEvent \leftarrow \langle qid, dist(s,b'), b', G_p, path(s,b'), ack\rangle$
18. $combiner[G'_p][b] \leftarrow rEvent$
19. $T \leftarrow CombineByPartition(combiner)$
20. **else if** $s \notin G_p$ && $t \notin G_p$ **then**
21. **if** $dist(s,b) < distCache(s,b)$ **then**
22. **for** each pair $\langle b', G'_p\rangle \in L_n$ **do**
23. $dist(s,b') \leftarrow dist(s,b) + dist(b,b')$
24. **if** $dist(s,b') < \delta_{s\to t}$ **then**
25. initialize a *rEvent*
26. $rEvent \leftarrow \langle qid, dist(s,b'), b', G_p, path(s,b')\rangle$
27. $combiner[G'_p][b] \leftarrow rEvent$
28. $T \leftarrow CombineByPartition(combiner)$
29. **else if** $s \notin G_p$ && $t \in G_p$ **then**
30. **if** $dist(s,b) < distCache(s,b)$ **then**
31. dist(b,t) $\leftarrow$ Dijkstra(b,t)
32. **if** dist(s,b)+dist(b,t)$< \delta_{s\to t}$ **then**
33. initialize a *PathEvent pEvent*
34. $pEvent$.path $\leftarrow path(s,b) \bowtie path(b,t)$
35. $\delta_{s\to t} \leftarrow dist(s,b) + dist(b,t)$
36. broadcast $\delta_{s\to t}$ to all the other *ShortPathPE*
37. emit *pEvent* to *StatPE*
38. $T \leftarrow T \bigcup \langle qid, G_p, G_p, ack\rangle$
39. $T \leftarrow T \bigcup \langle qid, event.nei, G_p, event.ack\rangle$
40. $sEvent.tuples \leftarrow T$
41. emit *sEvent* to *StatPE*

In Figure 2, we combine the *RouteEvent*s between G_1 and G_2 as one message and then send it to G_2.

5.2.2 Message Broadcast

Another optimization technique to avoid message flooding is to broadcast the shortest path to all the *ShortPathPE*s whenever a better result is found at the target partition G_t. In this way, each partition G_p maintains a variable $\delta_{s\to t}$, indicating the current best distance. The purpose is that the partitions far away from s can learn this information before receiving messages from neighbors. Then, the message propagation process can stop earlier without involving too many graph partitions. We define the distance from s to a graph partition as follows:

$$d(s,G_p) = \min_{v \in G_p.border} d(s,v) \quad (1)$$

If $d(s,G_p) > \delta_{s\to t}$, this propagation can stop. Although this condition may not be satisfied as the message transmission in the network is asynchronous, it can avoid spreading partial worse results from the source partition G_s to every other partition. The partition far away from s can dismiss the events if the partial result is found to be worse than the broadcasted result (lines 10 and 36 in Algorithm 1). Although the broadcast overhead is not cheap (the number of messages is the same as that of graph partitions), it still gains an advantage compared to the overhead caused by the communication between any two neighboring *ShortPathPE*s far from s.

If a graph partition G_p does not receive any *RouteEvent* from its neighbors, we have $d(s,G_p) \geq d(s,t)$. That means this partition is far away from source vertex s and the shortest path between s and t must not pass through this partition. The correctness and feasibility are given in Appendix A.1, A.2, A.3. This property can facilitate the pruning procedure in answering a RSP query in Section 6.

5.3 Algorithm Termination Mechanism

Algorithm termination is a critical issue in distributed query processing. For example, Pregel terminates when every vertex `votes to halt` in each superstep. In this paper, a solution is proposed for CANDS to determine algorithm termination. We create a special PE, named *StatPE*, to receive statistics of messages sent and received (which are encapsulated in *StatEvent*) from *ShortPathPE* and determine when query processing can be terminated. A *StatEvent* contains multiple tuples in the form of $\langle qid, G_p, G'_p, ack\rangle$, each acknowledges the communication with its neighboring partition. It means the message is sent from G_p to G'_p with acknowledgement sequence *ack*. Two tuples T_1 and T_2 match each other if the following condition is satisfied:

$$T_1.G_p = T_2.G_p \wedge T_1.G'_p = T_2.G'_p \wedge T_1.ack = T_2.ack$$

In *StatPE*, we maintain a buffer and a match counter for each query. When a *StatEvent* arrives, we scan the tuples in the event. For each tuple, we scan the buffer to find if there is a match. If a match is found, the tuple is removed from the buffer and the match counter increases by 1. Otherwise, we insert the tuple into the buffer. The algorithm is shown in Algorithm 2. It can be terminated as long as the buffer is empty and $match > 0$. All PEs can fully utilize the CPU resources without being idle. The algorithm can be terminated immediately when the buffer is empty and $match > 0$. There is no need to vote for halt. We prove the correctness of our algorithm in Appendix A.4, A.5, A.6, A.7. Note that when there is message loss, the termination condition will not be satisfied. To avoid endless waiting, we set a threshold for maximum waiting time for early termination of a failed query processing.

Algorithm 2 *StatEvent* handling algorithm in *StatPE*

1. **for** each tuple T in *StatEvent* **do**
2. **for** each tuple T' in buf w.r.t to $Q_{s\to t}$ **do**
3. **if** T matches T' **then**
4. remove T' from buf
5. $match \leftarrow match + 1$
6. **if** buf is empty and $match > 0$ **then**
7. emit result to *OutputPE*

6. REVERSE SHORTEST PATH QUERY PROCESSING

In this section we present the incremental update component shown in Figure 1. The traffic conditions are estimated in real-time based on the sensor data from vehicles. Whenever the status of a road changes, the shortcuts calculated previously in the corresponding partition needs to be updated, and a RSP query process is

triggered to find affected vehicles. If a better shortest path is found, the vehicles will be notified instantly. Since a shortest path can be affected by multiple updates at the same time, concurrency control is also required to avoid writing conflicts on the local caches that may lead to a wrong result.

6.1 Shortcut Update Algorithm

When the state of a road segment e at partition G_e changes, we need to update the shortcuts maintained for border nodes in G_e. A naive solution is to re-calculate the shortest path for each pair of border nodes, which is quite expensive. To reduce the maintenance overhead of shortcuts, different strategies are adopted in terms of the weight change and whether e appears in the original shortest path between border nodes b and b'.

- **Case 1**: $w_e \uparrow \ \wedge \ e \in P_{b\to b'}$ (lines 4-6). The shortest path between b and b' may not be optimal and we call Dijkstra algorithm to re-compute the shortest path.
- **Case 2**: $w_e \uparrow \ \wedge \ e \notin P_{b\to b'}$. The original shortcut is not affected by the update and is still optimal. No action is required.
- **Case 3**: $w_e \downarrow \ \wedge \ e \in P_{b\to b'}$ (lines 8-9). The shortcut between b and b' is still optimal. We simply need to update its distance because w_e decreases.
- **Case 4**: $w_e \downarrow \ \wedge \ e \notin P_{b\to b'}$ (lines 10-11). In this case, it is possible for border nodes b and b' to find a better result which passes edge e. Thus, Dijkstra algorithm is called to guarantee the shortcut is optimal.

6.2 RSP Query Processing

When a RSP query is triggered, we know that the state of an edge e changes. Our goal is to find which active queries are affected. A baseline solution adopts a similar strategy to the shortcut update algorithm that takes into account four cases. It re-computes the shortest path for case 1 and case 4 by submitting new navigation requests. For case 2 and case 3, the shortest paths remain the same and the vehicle is not affected. However, such a method is not scalable because a state update may trigger a large number of SSSP queries. We propose a more efficient method which takes advantage of intermediate partial routes produced in SSSP query processing. Partial route information will be cached until the related vehicle reaches its destination. When the state of a road segment e changes, we first traverse all active queries and detect which of them are affected. For each query $Q_{s\to t}$ with current location at s', we know that the original shortest path from s' to t is embedded in $s \to t$. In other words, we can obtain $P_{s'\to t}$ directly from $P_{s\to t}$. Similar to the baseline algorithm, we check whether the weight of e increases or decreases and whether it appears in the shortest path from s' to t. The query processing approach for case 2 and case 3 is the same as that in Section 6.1. In the following, we focus on how to efficiently handle cases 1 and 4.

6.2.1 Case 1: $w_e \uparrow \ \wedge \ e \in P_{s'\to t}$

To determine whether a navigation query $Q_{s\to t}$ with current location s' is affected when w_e increases and $e \in P_{s'\to t}$, we propose a backward message propagation method. The purpose of this traversal is to find a shortest path that does not pass through edge e and is shorter than existing $P_{s'\to t}$. If not, the existing path is still the shortest path. An *UpdateEvent*, which augments the fields in *RouteEvent* with updated edge e and current location s', is sent to the target partition G_t. The sketch of event processing is illustrated in Algorithm 3. When G_t receives the update event, it checks whether s' is contained in G_t. If so, Dijkstra algorithm is called to re-compute the shortest path (lines 3-7). Otherwise, the reverse traversal can start at t and pass through border node $b \in G_t$ to other partitions. The distance $dist(t,b)$ is calculated by Dijkstra algorithm. If $e \notin P_{s'\to b}$ [3], and $dist(s',b) + dist(t,b)$ is smaller than $\delta_{s'\to t}$, this is a potential shortest path (lines 9-14). If $e \in path(s',b)$, the border b continues to traverse in a reverse order and propagates it backward to its neighboring partitions (line 16).

In summary, our algorithm traverses in a reverse order from the target partition G_t if the cached path contains edge e. This algorithm only traverses the affected regions of edge e. The affected regions of edge e are the areas containing the vertex, whose shortest path passes through edge e from source vertex s. It stops propagating messages if it does not pass through edge e. We can find that the message number is related to the number of affected regions of this edge e. We take full advantage of cached paths and distances in the border to reduce the message transmissions. Our algorithm can quickly terminate when there is no message transmission in the network, as proved in Appendix A.8. Therefore, compared to the query processing algorithm that re-computes the shortest path, our approach incurs much less communication overhead.

Algorithm 3 BackwardMsgPropagation(*UpdateEvent event*)

```
1.  if t ∈ G_p then
2.    if s ∈ G_p then
3.      dist(s',t) ← Dijkstra(s',t)
4.      if dist(s',t) < δ_{s→t} then
5.        δ_{s'→t} ← dist(s',t)
6.        report δ_{s'→t} to StatPE
7.        broadcast δ_{s'→t} to all the ShortPathPE
8.  else
9.    if path(s',b) not pass e then
10.     dist(s',t) ← path(s',b) + dist(t,b)
11.     if dist(s',t) < δ_{s'→t} then
12.       δ_{s'→t} ← dist(s',t)
13.       report δ_{s'→t} to StatPE
14.       broadcast δ_{s'→t} to all the ShortPathPE
15.   else
16.     propagate event backward to the previous partition along
        path(s',b)
```

6.2.2 Case 4: $w_e \downarrow \ \wedge \ e \notin P_{s'\to t}$

When the weight of edge e decreases and e does not appear in the shortest path $P_{s'\to t}$, we propose a different solution from case 1 to detect the affected queries. The purpose of this traversal is to find a shortest path that passes through edge e and is shorter than existing $P_{s'\to t}$. If not, the existing path is still the shortest path. The algorithm handling case 4 is based on forward propagation and shown in Algorithm 4. Initially, an *UpdateEvent* is initialized and sent to the partition G_e containing e whose weight is updated. We traverse all cached queries in G_e. If a query $Q_{s\to t}$ (s is the initial source vertex and current location of that vehicle is s') is not cached in G_e, according to Lemma A.3, we know that the $dist(s,G_e) > \delta_{s\to t}$. Since $dist(s,G_e) \le \delta_{s\to s'} + dist(s',G_e)$ and $\delta_{s\to t} = \delta_{s\to s'} + \delta_{s'\to t}$, we have $dist(s',G_e) > \delta_{s'\to t}$. Thus, it is impossible to find a better result passing through e for $Q_{s'\to t}$. For each cached query, if G_e is its destination partition, we re-calculate the distance from the incoming border node b to t. Then, we notify *StatPE* and broadcast the new distance when a better result is found (lines 2-11). If t is not contained in G_e, from the cache we can fetch all the

[3]The cache contains path $P_{s\to b}$ and we can derive path $P_{s'\to b}$ from $P_{s\to b}$.

Algorithm 4 ForwardMsgPropagation(*UpdateEvent event*)

```
1.  for Query_{s'→t} cached in G_e do
2.    if t ∈ G_e then
3.      if s ∈ G_e then
4.        b ← s'
5.      else
6.        b is the incoming border node
7.      dist(b,t) ← Dijkstra(s,t)
8.      if dist(s',b) + dist(b,t) < δ_{s'→t} then
9.        δ_{s'→t} ← dist(s',t)
10.       report δ_{s'→t} to StatPE
11.       broadcast δ_{s'→t} to all the ShortPathPE
12.   else
13.     for each outgoing border node b' for query Q_{s→t} do
14.       if e is contained in the updated shortcut (b,b') then
15.         for Query_{s'→t} which passes b and b' do
16.           dist(s',b') ← dist(s',b) + dist(b,b')
17.           emit a RouteEvent with partial result dist(s',b') to
              neighboring partition through border node b'
```

outgoing border nodes b' to the neighboring partition. If the updated shortcut between b and b' passes through e, we create a new *RouteEvent* with the better distance $dist(s,b_{i_j})$ and emit the event to *ShortPathPE* in the query processing. The following processing of the new generated *RouteEvent* is the same as that in Section 5 and omitted here. The algorithm will terminate when there is no message in transit. Finally, we prove that Algorithm 4 finds all the affected queries in case 4 and can correctly update their shortest paths in Appendix A.9.

6.3 Concurrency Control

In CANDS, when there is a weight update in an edge, a reverse shortest path query is triggered and all the affected queries will be updated. Since an active navigation query could be affected by multiple edge update at the same time, concurrency control mechanism should be adopted to guarantee the correctness of the result. Figure 4 shows a write conflict example. Suppose initially at timestamp T_1, the shortest path from A to D is $A \to B \to C \to D$ with distance 12. At T_2 ($T_2 > T_1$), the first edge update occurs and the weight of AB increases from 3 to 5. Then, a reverse shortest path query U_1 is triggered to update all the affected navigation queries. However, before U_1 finishes, another edge update occurs and the weight of BC decreases from 4 to 2 at timestamp T_3 ($T_3 > T_2$). It triggers another query processing U_2 for edge update of BC. It is possible that U_2 finishes earlier than U_1 and update the distance $\delta_{A\to D}$ maintained in each processing element to be 10. Finally, at timestamp T_4 ($T_4 > T_3$), U_1 finishes. However, it does not take into account the update of U_2 and $\delta_{A\to D}$ is updated to 14. But the real shortest distance from A to D should be 12 at timestamp T_4. To solve the problem, we use a simple lock mechanism to guarantee that the query processing for edge update is executed sequentially. More specifically, when there is an edge update, we first check whether the lock is occupied by other update events. If yes, the event is pushed into a waiting queue. Otherwise, it gets the lock, processes the edge update event and releases the lock. In this manner, we can avoid a navigation query being affected by multiple concurrent edge updates. It is worth noting that all our experiments are conducted with such locking mechanism.

7. EXPERIMENTS

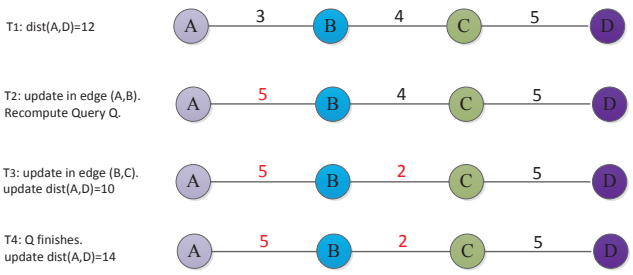


Figure 4: Write conflict when there is no concurrency control

We implement CANDS on S4 [2], which is a general-purpose distributed stream processing system under Apache incubation. S4 provides friendly programming interfaces and supports an unbounded stream. This system is event driven and query processing in each PE can be triggered periodically or by an incoming event. The version of Apache S4 is 0.6.0. It is worth noting that although we have deployed CANDS on S4, CANDS can also be implemented on other stream systems such as Twitter Storm [7]. All of our experiments are run on a cluster with ten nodes. Each node has one Xeon E5607 Quad Core CPU (2.27GHz), 32GB memory, running CentOS 6.2. We select one node for Zookeeper and data source adapter in the customization of S4.

We study the performance of SSSP and RSP query processing in CANDS. For SSSP query, we report the results of our asynchronous algorithms without optimization (Async), improved with *Message Broadcast* only (Async+MB), improved with *Message Combiner* only (Async+MC), and using both optimization techniques (Aysnc+MB+MC). We compare performances of CANDS and GPS (Graph Processing System), an open source implementation of Google's Pregel. For RSP query processing, we compare with a baseline algorithm which re-computes an SSSP query if the query is detected to be affected by the status update of an edge. We measure the performance of SSSP and RSP query processing from the following four aspects:

Average query latency. The latency of an SSSP query is the time interval between the time it arrives at the system and the time it has been processed. For RSP query, the latency incorporates the time waiting for the release of the lock of concurrency control and the query processing time, which refers to the time for detecting all the affected queries and updating them with better routes. If a query fails, we do not count it when calculating the average query latency.

Throughput. Throughput is measured by the number of queries processed in a time unit. In our setting, the time unit is one minute.

Network I/O. The network I/O is measured by the amount of bytes of messages transferred in the network, which is the same metrics to measure the communication cost in GPS.

Success rate. Since messages may be lost when the message arrival rate is too high, we use the query success rate as a measurement of system availability. For SSSP query, it is the number of successful queries divided by the total number of queries. For RSP query, we consider it successful only if all the affected navigation requests are updated with new correct results. Note that this measures the success rate for first time processing and we have fault tolerance for failed query(e.g., re-submit the query).

In the following, we report performance evaluation using both synthetic and real workloads, and all experiments are conducted with concurrency control described in Section 6.3.

7.1 Experiments with Synthetic Workloads

7.1.1 Datasets and Experiment Setups

Our experiments use datasets derived from US road network [3]. We pick 5 representative road networks of different sizes and summarize them in Table 3. The smallest dataset contains only 320K vertices and the largest one contains more than 14 million vertices. By default, we use the CAL dataset with moderate size to test the performance.

Table 3: Road Network Datasets

Name	Region	Vertex Number	Edge Number
BAY	San Francisco Bay Area	321,270	800,172
FLA	Florida	1,070,376	2,712,798
CAL	California and Nevada	1,890,815	4,657,742
E	Eastern USA	3,598,623	8,778,114
CTR	Central USA	14,081,816	34,292,496

Table 4: Experiment Parameters

cluster nodes	1, 2, 3, 4, 5, 6, 7, 8, **9**
length of shortest path	[1,100], **[100,200]**, [200,300], [300,400], [400,500], [500,600]
graph partition	10, 100, 500, **1000**, 5000, 10000
update frequency	**200**, 400, 600, 800
active SSSP queries	10, 100, 500, **1000**, 5000, 10000
ratio of RSP query	0.1, **0.3**, 0.5, 0.7, 0.9

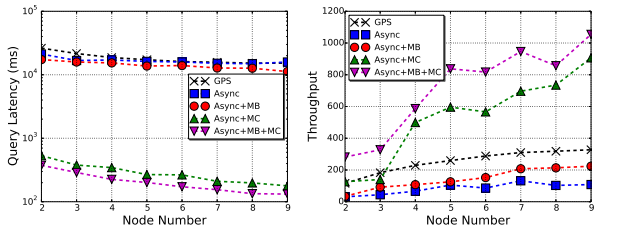
We evaluate the performance in terms of increasing numbers of nodes in a cluster, length of path route, number of graph partitions, frequency of edge updates and number of active queries. We also test the hybrid case where SSSP and RSP queries arrive together to indicate the ratio of RSP queries to the total number of query events. The parameters are listed in Table 4, with the default setting in bold format.

7.1.2 SSSP Query Processing

We first examine the performance of SSSP query in terms of increasing number of nodes in a cluster. The query latency, throughput, network I/O and success rate are reported in Figure 5. We compare with GPS and report the performance with different optimization techniques. When more nodes are used, it incurs more network I/O cost to process SSSP queries. However, the query latency and system throughput are both improved because we have more memory and CPU resources for parallel computing. The message combiner (MC) plays a more important role than message broadcast (MB) in the optimization of SSSP query processing. It is very effective in reducing network I/O and query latency. It improves system throughput by 5 times. MB method can reduce the communication cost, as shown in Figure 5(c), but the effect is limited.

GPS uses bulk synchronous parallel model and in each superstep, a vertex handles messages coming from its neighboring vertices. If there is no incoming message in the current iteration, it votes to halt. Therefore, given a query with length L (meaning the route has L edges), it requires at least L supersteps to finish a query. In Figure 5(a), the length of SSSP query results is between 100 and 200 and we report the performance of 100 supersteps in GPS. When deployed in 9 nodes, our system takes only 132ms to answer an SSSP query and supports more than 1000 queries in one minute without any failure. The query latency of our system is *two orders of magnitude* better than GPS.

The performance of SSSP query processing is also sensitive to the query result length. We increase the length from $[1,100]$ to $[500,600]$ and show the performance in Figure 6. The query latency grows with increasing path length. By comparing the query latency between Async+MC and Async+MC+MB, we can see that message combiner is more effective in performance improvement when the length increases. Async+MB has 100% success rate if the length is less than 200 and decreases when more than 200. The

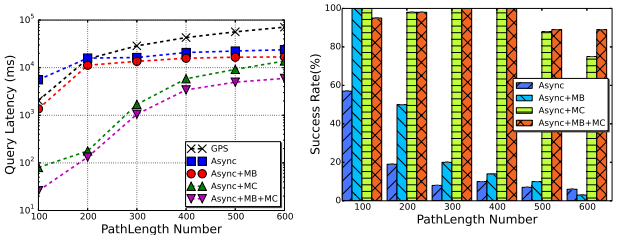


(a) Avg. query latency (b) Throughput

(c) Network I/O (d) Success rate

Figure 5: Performance w.r.t. increasing nodes in a cluster

success rates of Async+MC and Async+MB+MC are still very high even if the path length of SSSP query is very large (e.g., 500-600). Async+MB is more affected than Async+MC and Async+MB+MC by path length.



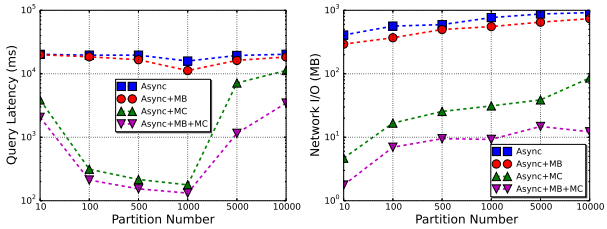
(a) Avg. query latency (b) Success rate

Figure 6: Performance w.r.t. increasing length of query result

The query latency and network I/O of SSSP query processing with increasing number of graph partitions are estimated in Figure 7. The communication I/O grows significantly when the number of partitions increases from 10 to 10,000. The query processing time first drops when the graph is split into more partitions. When the partition number is very small, Dijkstra algorithm is expensive to find the distance from source vertex s to border vertices or from border vertices to target vertex t. However, when the partition number grows to thousands, the performance degrades because it incurs high communication cost and much effort is spent in sending and receiving messages. Async+MB+MC always achieves the best performance as the partition number grows.

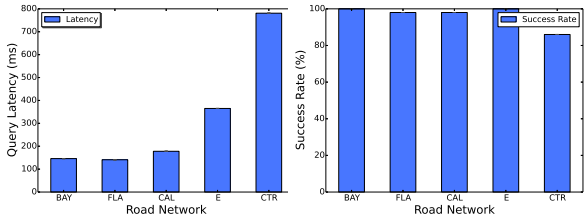
In the final experiment of SSSP query processing, we report the average query latency and success rate in different road networks in Figure 8. When the graph size increases, the PEs need to take charge of a larger graph partition and the performance of SSSP query processing degrades. In CTR, which contains 14 million vertices, the elapsed time for one query is more than 700ms. The success rate also drops when the graph size increases dramatically. But our system still achieves low query latency and high success rate in a larger graph.

7.1.3 RSP Query Processing



(a) Avg. query latency (b) Network I/O

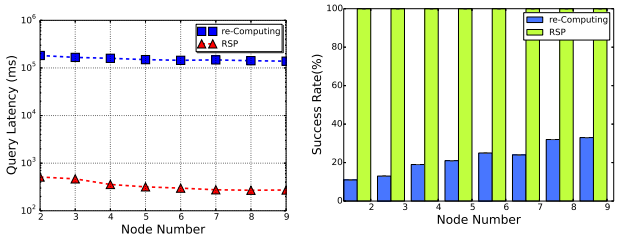
Figure 7: Performance w.r.t. increasing number of partitions



(a) Avg. query latency (b) Success Rate

Figure 8: Performance in different road networks

We compare the efficiency of query update between RSP query and re-computing query. Re-computing query represents a baseline approach that resubmits the query to find the new shortest path once the query is affected. The performance of RSP query processing is shown in Figure 9. The default number of active queries is 1000. When the number of nodes increases, the query latency drops slightly. The successful rate is always 100 percent. With 9 nodes, the average time to run an RSP query is 273ms. Our method utilizing the cached partial results achieves two orders of magnitude of performance improvement over the baseline algorithm which uses re-computation to update the affected queries.

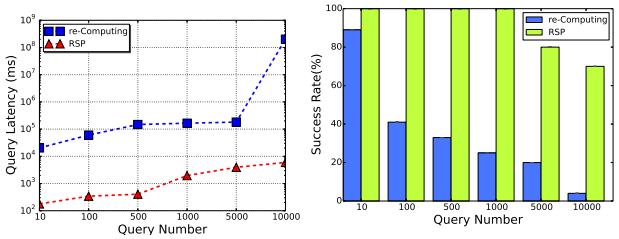


(a) Avg. query latency (b) Success rate

Figure 9: Performance w.r.t. increasing nodes in a cluster

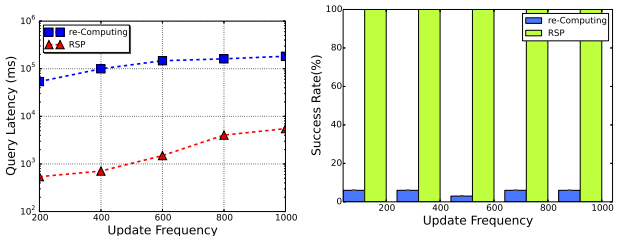
We report the average query latency and success rate with increasing number of active SSSP queries in Figure 10. With more active queries in the system, more of their shortest paths will be affected by an update of the road state. Therefore, it takes more time to complete an RSP query. When the number of active queries increases to 5000, it takes 4 seconds to detect and update affected queries and the success rate also drops significantly.

The update frequency is another aspect to estimate the performance of our system. When the weights of many edges are updated at same time, a lot of RSP queries may be affected. In Figure11 we present the average query latency and success rate with increasing update frequency. Our RSP queries can finish in one second when



(a) Avg. query latency (b) Success rate

Figure 10: Performance of RSP query w.r.t. increasing active queries

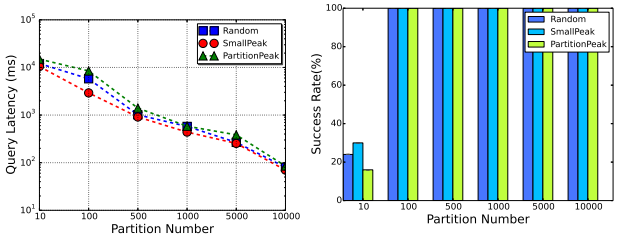


(a) Avg. query latency (b) Success rate

Figure 11: Performance w.r.t. increasing update frequency

update frequency is less than 500 per minute. The result shows that RSP processing can keep up with increasing number of edge updates.

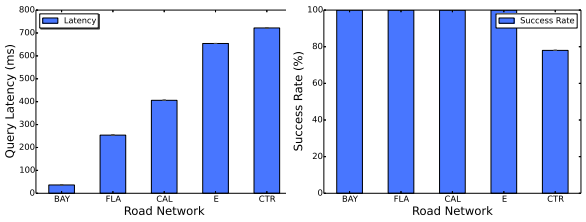
Since a traffic jam might affect many edges, the partition will be very busy to process related edges. It will affect the performance of our system. We apply three methods on choosing update edges. *Random* randomly chooses some edges to update with random values. *SmallPeak* selects update edges based on *Random* method, but the weights of some edges are updated with high values. *PartitionPeak* means some partitions are very busy and the edges in these partitions are peak. RSP queries can be finished in one second and 100 percent success rate when the number of partition is more than 500. In Figure 12, we estimate the performance of these three methods and our system can quickly process a series of weight update.



(a) Avg. query latency (b) Avg. Success rate

Figure 12: Performance w.r.t. Sampling and Peak Methods

The performance of RSP query processing on all five datasets is shown in Figure 13. If the size of road network is small, our method is very efficient with 100 percent success rate. It takes 36ms to process an RSP query in the BAY dataset. Even in the CTR road network which contains more than 14 million roads, the average response time to an RSP event is around 700ms. There is no query failure in these datasets except CTR. The result shows that RSP query can finish the query update in a short time even in a large road network.

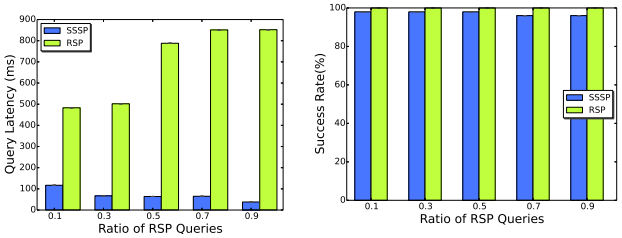


(a) Avg. query latency (b) Success rate

Figure 13: Performance in different road networks

7.2 Hybrid Query Event Processing

We control the ratio of RSP queries in a hybrid scenario where SSSP and RSP queries arrive in a mixed way. The ratio of RSP queries indicates the ratio of RSP queries among the total number of query events. The total number of queries is 300 and the ratio increases from 0.1 to 0.9. The average latency of SSSP query processing improves when the ratio becomes larger. As we can see from Figure 14, the query latency of RSP query processing increases as the ratio becomes larger. The reason is the concurrency control mechanism works when an SSSP query is affected by multiple road weight updates. Thus, it takes more waiting time when there are more RSP queries in the event stream. The success rate of query processing is nearly 100 percent when the ratio varies.



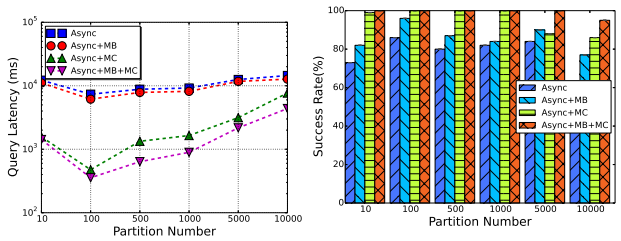
(a) Avg. query latency (b) Success rate

Figure 14: Performance w.r.t. to ratio of RSP query

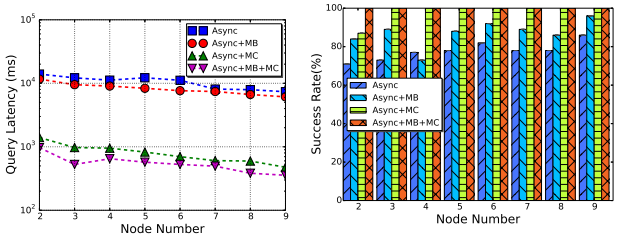
7.3 Experiments with Real Workloads

The real trace dataset contains the GPS trajectories of 10,357 taxis during the period of Feb. 2 to Feb. 8, 2008 within Beijing [27]. There are 154,662 vertices and 337,662 road segments in the Beijing road network. Each trajectory is a sequence of GPS records with timestamp and status information. These GPS records are mapped to the corresponding network edges and the original travelling path is inferred [25]. We split a day into multiple time intervals and for each interval, we estimate the weight of road network by the average travel time of all the taxies passing through the edge in that period. We extract the boarding and alighting points of passengers as s and t in query $Q_{s \to t|T}$. In this way, we generated 200 queries. The trace of these queries has an average of 200 path lengths.

The performance of SSSP query is evaluated with increasing partition number and node number. The query latency and success rate are reported in Figure 15. The best partition number is 100 and has higher success rate. The query latency of Async+MC and Async+MB+MC decreases with more nodes in all query methods, which is similar to that for *Synthetic Workloads*. The success rates of Async and Async+MB are higher than those for *Synthetic Workloads*.



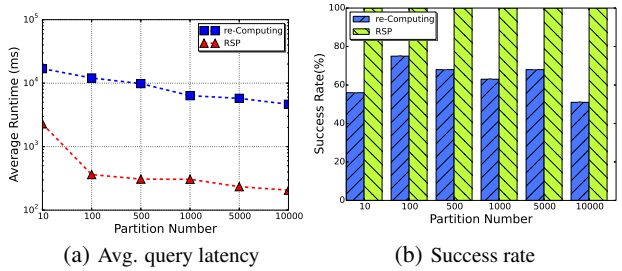
(a) Avg. query latency (b) Success rate



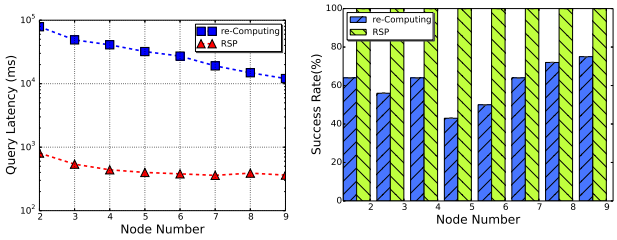
(c) Avg. query latency (d) Success rate

Figure 15: Performance Evaluation of SSSP query

We also evaluate the performance of RSP query on Beijing road network. The weight of the road is changing with different traffic conditions. We define four types of road statuses $(green, blue, yellow, red)$ to denote traffic condition with increasing congestion. The range of each status is based on the default travel time ω of this road. We define four ranges: $(0, 1.25*\omega)$ as *green*, $[1.25*\omega, 2*\omega)$ as *blue*, $[2*\omega, 3*\omega)$ as *yellow*, and $[3*\omega, \infty)$ as *red*. If the status of one road changes, RSP query will be triggered to find affected queries and recommend new shortest paths for these queries. We estimate the query latency and success rate between RSP query and re-computing method in Figure 16. RSP query has a faster and higher success rate result than re-computing method, which is similar to results in previous simulation.



(a) Avg. query latency (b) Success rate



(c) Avg. query latency (d) Success rate

Figure 16: Performance Evaluation of RSP query

8. RELATED WORK

Shortest path query processing has been well studied [23, 14, 8, 12, 22, 28]. Most of the recent works focus on developing index

structures to support query processing in a huge road network. For example, ALT [13] selects some vertices as *landmarks* and precomputes the distance to these landmarks to accelerate query processing. RE [14] adds shortcuts between pairs of selected vertices to help pruning. In TNR [10], grid network is adopted to reduce the search space. In [8, 12, 22, 28], the vertices are organized in a hierarchical structure and shortcuts between different levels of vertices are added. Although these indexes are efficient in answering shortest path queries, they require tremendous pre-computing cost to build the index and are not suitable for a dynamic road network.

There are various mining algorithms proposed to predict travel time across a road segment. For example, T-drive [26] mines smart driving directions from the historical GPS trajectories of a large number of taxis and constructs a time-dependent landmark graph to recommend the best route. Another adaptive algorithm [15] is proposed to estimate traffic condition from historical traffic data. *VTrack* [24] can estimate the travel time along the route based on the noisy information from different sources such as WiFi. IBM also implemented a real-time traffic estimation system based on IBM InfoSphere Streams [11, 5]. *MobileMillennium* [18], developed in Spark [6], is another system supporting traffic estimation and prediction. It infers traffic conditions using GPS measurements from drivers running cell phone applications, taxicabs, and other mobile and static data sources.

Distributed graph processing systems such as Pregel [20], GPS [4] and Giraph [1] are developed to efficiently process large-scale web graphs and various social networks. These systems require much effort in synchronization and are designed for general graph processing algorithms. Therefore, our customized method can achieve two orders of magnitude improved performance. Although GraphLab [16] also adopts asynchronous methods to support scalable graph mining in an asynchronous manner, its current implementation only works in a static graph.

In response to traffic delay update, Malviya [21] proposed two approximate techniques, *K-paths and proximity measure*, to monitor a set of designated travel routes. Our system has two advantages compared to this work. We return exact results and notify all the affected vehicles when there is a traffic update.

9. CONCLUSION

In this paper, we studied the problem of the shortest path over a dynamic road network and implemented the service on a distributed stream processing platform S4. More specifically, we proposed an asynchronous and optimized method to efficiently compute shortest path over a dynamic graph. To guarantee the optimality in the navigation process, we monitor the traffic updates in real-time and notify affected vehicles if a better route is found. Experiments on real road networks with synthetic traffic loads and real trajectory trace verified the efficiency of our system. It takes less than 150ms to respond to a navigation request in a network with millions of vertices if the route length is moderate. It also supports $10,000$ active navigation queries without failure when identifying affected queries by a status update of any road segment. Our system can achieve low query latency asynchronous query processing and fast response time on query updates.

10. ACKNOWLEDGEMENTS

The work of Kian-Lee Tan and Dongxiang Zhang are funded by the NExT Search Centre (grant R-252-300-001-490), supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office. This work is also partially supported by China NSF(Grant No. 61272438), Research Funds of Science and Technology Commission of Shanghai Municipality (Grant No. 14511107702, 12511502704). Frédéric Le Mouël's work is funded by a grant from Rhone-Alpes Region, France.

11. REFERENCES

[1] Apache Giraph. http://giraph.apache.org/.
[2] Apache S4. http://incubator.apache.org/s4/.
[3] DIMACS. http://www.dis.uniroma1.it/challenge9/download.shtml/.
[4] GPS. http://infolab.stanford.edu/gps/.
[5] Infosphere Streams. http://www-01.ibm.com/software/data/infosphere/streams/.
[6] Spark. http://spark.incubator.apache.org/.
[7] Twitter Storm. https://github.com/nathanmarz/storm.
[8] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA*, pages 782–793, 2010.
[9] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360, 2013.
[10] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566–566, 2007.
[11] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. Ibm infosphere streams for scalable, real-time, intelligent transportation services. In *SIGMOD*, pages 1093–1104. ACM, 2010.
[12] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*, pages 319–333. Springer, 2008.
[13] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, pages 156–165, 2005.
[14] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In *ALENEX*, volume 6, pages 129–143, 2006.
[15] H. Gonzalez, J. Han, X. Li, M. Myslinska, and J. P. Sondag. Adaptive fastest path computation on a road network: a traffic mining approach. In *VLDB*, pages 794–805. VLDB Endowment, 2007.
[16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.
[17] A. Guerrero-Ibáñez, C. Flores-Cortés, P. Damián-Reyes, M. Andrade-Aréchiga, and J. Pulido. Emerging technologies for urban traffic management. Technical report, 2012.
[18] T. Hunter, T. M. Moldovan, M. Zaharia, S. Merzgui, J. Ma, M. J. Franklin, P. Abbeel, and A. M. Bayen. Scaling the mobile millennium system in the cloud. In *SOCC*, page 28, 2011.
[19] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
[20] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.
[21] N. Malviya, S. Madden, and A. Bhattacharya. A continuous query system for dynamic route planning. In *ICDE*, pages 792–803, 2011.
[22] M. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *VLDB*, 4(2):69–80, 2010.
[23] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, pages 568–579. Springer, 2005.
[24] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson. Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *SenSys*, pages 85–98. ACM, 2009.
[25] H. Wei, Y. Wang, G. Forman, Y. Zhu, and H. Guan. Fast viterbi map matching with tunable weight functions. In *SIGSPATIAL GIS*, pages 613–616. ACM, 2012.
[26] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *SIGSPATIAL GIS*, pages 99–108. ACM, 2010.
[27] Y. Zheng, Y. Liu, J. Yuan, and X. Xie. Urban computing with taxicabs. In *Ubicomp*, pages 89–98, 2011.

[28] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *SIGMOD*, pages 857–868, 2013.

APPENDIX

A. THE PROOF OF LEMMA

Here we prove that the cached distance in any partition G_p is more than the shortest path $d(s,t)$ if this partition does not receive any *RouteEvent*.

LEMMA A.1. *For any partition G_p, its cached $\delta_{s\to t} \geq d(s,t)$.*

PROOF. If the partition does not receive a broadcast message from the destination PE, $\delta_{s\to t}$ is initialized to ∞ and thus $\delta_{s\to t} \geq d(s,t)$. Otherwise, we have found a path from s to t with distance $\delta_{s\to t}$. Since $d(s,t)$ is the distance of the shortest path, $\delta_{s\to t} \geq d(s,t)$. □

LEMMA A.2. *For any partition G_p, if $d(s,G_p) < d(s,t)$, G_p receives at least one RouteEvent from its neighbors.*

PROOF. Assume that the shortest path from s to G_p is $s \to \ldots u \to v$ where $u \in G'_p$, $u \notin G_p$ and $v \in G_p$. In other words, G'_p is the neighboring partition of G_p with a connecting edge (u,v). We know that in G'_p, we have $dist(s,u)+dist(u,v) = dist(s,G_p) < d(s,t)$. From Lemma A.1, we have $d(s,t) \leq \delta_{s\to t}$. Therefore, $dist(s,u)+dist(u,v) < \delta_{s\to t}$ and a *RouteEvent* will be sent from G'_p to G_p based on Algorithm 1. □

LEMMA A.3. *If a graph partition G_p does not receive a message w.r.t. to $Q_{s\to t}$ from its neighbor, we have $d(s,G_p) \geq d(s,t)$.*

PROOF. It is easy to prove from LemmaA.2. □

Now we prove that if $match > 0$ and $buf = \emptyset$, the algorithm can be terminated with correct shortest path.

LEMMA A.4. *Given a message propagation chain from $G_1 \to G_2 \to \ldots \to G_n$, if a set of acknowledgements from $\{G_{j_1}, G_{j_2}, \ldots\}$ arrives earlier than G_i where $j_k > i$, we can find at least one acknowledgement from G_{j_k} $(j > i)$ pending in the buffer.*

PROOF. Let $j_u = \min\{G_{j_1}, G_{j_2}, \ldots\}$. Since the acknowledgement of $G_{j_{u-1}}$ has not arrived, the acknowledgement from G_{j_u}, $\langle qid, G_{j_{u-1}}, G_{j_u}, ack\rangle$ cannot find a match and will be inserted into the buffer as in Algorithm 2. □

LEMMA A.5. *If $match > 0$ and $buf = \emptyset$, all the RouteEvents have been processed.*

PROOF. Suppose at timepoint t, we have $match > 0$ and $buf = \emptyset$ in *StatPE* and there still exists one *RouteEvent* sending from G_p to G'_p that has not been processed. The event could be either under transmission in the network or queuing in the event buffer in G'_p. We assume that the partial result from s to G'_p in this message is $s \to v_1 \to v_2 \to \ldots \to u \to v$ and this path crosses a set of partitions $G_1 \to G_2 \to \ldots \to G_n$. If there exists no partition G_i from which *StatPE* has received a tuple $\langle qid, G_i, G_{i+1}, ack\rangle$ used to acknowledge its sending a *RouteEvent* to the neighbor G_{i+1} in this path. According to Lemma A.4, the buffer will not be empty; If *StatPE* has received an acknowledgement from G_i. Suppose G_i is closest to G_n in this chain and the acknowledgement G_i has been sent to *StatPE*. Since the acknowledgement from G_{i+1} has not arrived, the one from G_i cannot be matched and it will be pending in the buffer. This contradicts the fact that the buffer is empty. □

LEMMA A.6. *If $match > 0$ and $buf = \emptyset$ in StatPE, the algorithm can terminate with the correct shortest path from s to t.*

PROOF. From Lemma A.5, we know that when $match > 0$ and the message buffer in *StatPE* is empty, all the *RouteEvent*s have been processed and acknowledged and there is no need for the *StatPE* to continue waiting for messages coming from *ShortPathPE*. The algorithm can terminate.

Let P be the real shortest path and $P = (s \to v_{s_1} \ldots) \to (v_{11} \to v_{12} \ldots) \to (v_{21} \to v_{22} \ldots) \to \ldots \to (v_{m1} \to \ldots \to v_t)$ which passes a sequence of m graph partitions $(G_1) \to (G_2) \to \ldots \to (G_m)$. Support a node v_{ij} on the shortest path, that is $d(s,t) = d(s,v_{ij}) + d(v_{ij},t)$, if there exists a new path from s to v_{ij}, $d'(s,v_{ij}) < d(s,v_{ij})$. It will send the new path with the partial result to neighbor partitions which leads to a contradiction as there is no message in the network. Therefore, our algorithm finds the optimal shortest path from s to t. □

Similar to Lemma A.6, we can prove the correctness of the following lemma:

LEMMA A.7. *When the algorithm terminates, for any border node b, if $dist(s,b) < \delta_{s\to t}$, then $dist(s,b)$ is the shortest distance from s to b.*

Here we prove Algorithm 3 and Algorithm 4 can efficiently handle the edge update on Case 1 ($w_e \uparrow \ \wedge \ e \in P_{s'\to t}$) and Case 4 ($w_e \downarrow \ \wedge \ e \notin P_{s'\to t}$) respectively.

LEMMA A.8. *Algorithm 3 on Case 1 is correct.*

PROOF. Suppose $P = s' \to u_1 \to \ldots \to u_i \to u_{i+1} \to \ldots t$ is the old shortest path passing the update edge $e = (u_i, u_{i+1})$ and $P' = s' \to v_1 \to v_2 \to \ldots \to t$ is a new shortest path with a smaller distance. It is obvious that e does not appear in the new path P'. If s', v_i and t are in the same partition, the shortest path will be returned by calling Dijkstra algorithm (lines 3-7 in Algorithm 3). Otherwise, we split P' into two segments $P' = (s' \to \ldots v_i) \to (v_{i+1} \ldots \to t)$, where v_i is the first node in the backward direction of path P' such that $P_{s'\to v_i}$ in the cache of *ShortPathPE* does not contain edge e. Otherwise, all the v_i in path P' whose cached path $P_{s'\to v}$ contains edge e. It means $e \in P'$, which is a contradiction. According to Lemma A.7, since the cached $dist(s',b) < \delta_{s'\to t}$, we know that $dist(s',b)$ is the shortest distance from s' to b. Since the cached partial results from s' to $v_{i+1}, v_{i+2}, \ldots, t$ contain edge e, the partition containing v_i will receive a backward propagation (line 16 in our algorithm). Thus, it will find a better result by merging the cached shortest path from s' to v_i and the propagated partial result from neighboring partition (lines 9-14). □

LEMMA A.9. *Algorithm 4 on Case 4 is correct.*

PROOF. Suppose $P' = s' \to v_1 \to \ldots (v_j \to \ldots v_i \to v_{i+1} \ldots) \to \ldots \to t$ is a new shortest path and $\{v_j, \ldots, v_i, v_{i+1}\} \in G_e$ where v_j is the incoming border node and (v_i, v_{i+1}) is the edge with weight update. If $G_e = G'_s$, we consider $v_j = s'$. According to Lemma A.7, the cached distance $dist(s',v_j)$ in G_e is the shortest distance from s' to v_j in P'. If $t \in G_e$, we call Dijkstra algorithm to calculate the shortest path between v_j and t. It is easy to show that the distance of $path(v_j,t)$ is the same as that of $v_j \to \ldots \to t$ in P'. Thus, a shortest path with the same distance as P' will be reported by Algorithm 4. If $t \notin G_e$, suppose v'_j is the outgoing border node of G_e in path P'. Since the shortcut (v_j, v'_j) passes the update edge e, we create a *RouteEvent* which contains the partial result $path(s',v'_j)$ and propagate it to the neighboring partitions. Similar to the proof in Lemma A.6, we can show that P' will be found by our query processing algorithm. □