Highly Scalable Trip Grouping for Large–Scale Collective Transportation Systems

Gyozo Gidofalvi
Geomatic ApS
gyg@geomatic.dk

Torben Bach Pedersen
Aalborg University
tbp@cs.aau.dk

Tore Risch
Uppsala University
tore.risch@it.uu.se

Erik Zeitler
Uppsala University
erik.zeitler@it.uu.se

ABSTRACT

Transportation–related problems, like road congestion, parking, and pollution, are increasing in most cities. In order to reduce traffic, recent work has proposed methods for vehicle sharing, for example for sharing cabs by grouping "closeby" cab requests and thus minimizing transportation cost and utilizing cab space. However, the methods published so far do not scale to large data volumes, which is necessary to facilitate large–scale collective transportation systems, e.g., ride–sharing systems for large cities.

This paper presents highly scalable *trip grouping algorithms*, which generalize previous techniques and support input rates that can be orders of magnitude larger. The following three contributions make the grouping algorithms scalable. First, the basic grouping algorithm is expressed as a continuous stream query in a data stream management system to allow for a very large flow of requests. Second, following the divide–and–conquer paradigm, four space–partitioning policies for dividing the input data stream into sub–streams are developed and implemented using continuous stream queries. Third, using the partitioning policies, parallel implementations of the grouping algorithm in a parallel computing environment are described. Extensive experimental results show that the parallel implementation using simple adaptive partitioning methods can achieve speed–ups of several orders of magnitude without significantly degrading the quality of the grouping.

1. INTRODUCTION

Transportation–related problems, like congestion, parking, and pollution, are increasing in most cities. Waiting in traffic jams not only degrades the quality of social life, but according to estimates, the economic loss caused by traffic jams in most countries is measured in billions of US dollars yearly. Parking is also a serious problem. In some large cities, it is estimated that as many as 25% of the drivers on the road are only looking for empty parking places. This again causes unnecessary congestion. Finally, the increasing number of vehicles idling on the roads results in an unprecedented carbon emission, which has unquestionably negative effects on the environment.

By reducing the number of vehicles on the roads, Collective Transportation (CT) clearly provides a solution to these problems. Public transportation, the most common form of CT, tries to meet the general transportation demands of the population at large. By generalizing the transportation needs, the individual is often inconvenienced by long wait times at off–peak hours or between connections, and a limited number of access points (bus, metro, train stops) from which the individual is forced to use other methods of transportation (walking, bicycling, using a private car). Ride–sharing, or car pooling, which is another form of CT is becoming widespread in metropolitan areas. Ride-sharing is often encouraged by local transportation authorities by facilitating car pool lanes that are only accessible to multiple–occupancy vehicles and by eliminating tolls on bridges and highways for these vehicles. Despite all the encouragement, there is a tremendous amount of unused transportation capacity in the form of unoccupied seats in private vehicles. This fact can mainly be attributed to the lack of effective systems that facilitate large–scale ride–sharing operations. The systems that do exist [3, 15, 22] are either 1) offered from a limited number access points due to the system infrastructure constraints, 2) have inadequate methods for the positioning of trip requests and/or vehicles, or 3) have either inefficient or ineffective methods for matching or grouping trip requests and trip offers.

Yet another form of CT, namely cab–sharing, was recently proposed [12]. The key idea of cab–sharing is to use unoccupied cab space to reduce the cost of transportation, ultimately resulting in direct savings to the individual. The described Cab–Sharing System (CSS) overcomes most of the above limitations of existing ride–sharing systems. In particular, at the heart of the system is a trip grouping algorithm that is able to find subsets of closeby trip requests, which can be grouped into collective cab fares to minimize the transportation cost, or equivalently maximize the savings to the user. Using a simple implementation in standard SQL, assuming a reasonable number (high spatio–temporal

EDBT'08, March 25–30, 2008, Nantes, France.

density) of trip requests, the trip grouping algorithm was demonstrated to be able to group trip requests effectively. The trip grouping algorithm can be generalized to facilitate other CT systems, e.g., a ride–sharing system. However, as it is demonstrated in the present paper, due to its algorithmic complexity, the grouping algorithm scales poorly as the volume of trip requests increases. This limits its applicability to facilitate large–scale CT systems, such as a metropolitan or nation–wide ride–sharing system.

To make the trip grouping algorithm scale to input rates several orders of magnitude larger than in a typical cab-sharing application, this paper makes the following three contributions. First, using a Data Stream Management System (DSMS), SCSQ [24], the trip grouping algorithm is expressed as a continuous stream query to allow for continuous processing of large trip request streams. Second, following the divide–and–conquer paradigm, *static* and *adaptive* versions of two space–partitioning policies (*point quad* and *KD partitioning*) for dividing the input data stream into sub–streams are developed and implemented using continuous stream queries. Finally, using the partitioning policies, the grouping algorithm is implemented using a data stream management system in a parallel computing environment. The parallelization of the implementation is facilitated by using an extension of the query language, in which processes are query language objects. Extensive experimental results show that the parallel implementation using simple partitioning methods can achieve speed–ups of several orders of magnitude without significantly affecting the quality of the grouping. In particular, an adaptive partitioning method called *adaptive KD partitioning* achieves the best overall performance and grouping quality.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 defines the vehicle–sharing problem, reviews the operational aspects of a recently proposed Cab–Sharing System (CSS), describes and analyzes a trip grouping algorithm that solves the vehicle–sharing problem and is employed to facilitate the CSS. Furthermore, a new Ride–Sharing System (RSS) is proposed, and the trip grouping algorithm is adapted to meet the application requirements of the proposed RSS. Section 4 describes the main contributions of the paper in making the trip grouping algorithm highly scalable, hence applicable in large–scale CT system, such as an RSS. Section 6 describes and analyzes the results of the experiments that were conducted to measure the performance of the proposed highly scalable trip grouping algorithm. Finally, Section 7 concludes and points to future research directions.

2. RELATED WORK

The optimization of CT has been studied in the scientific community for years [5, 20]. However, with the exception of the work presented in [12], on which the present paper is based, it is believed that no previous research has considered the online grouping of trip requests. The problem of grouping n objects into a number of groups is in general referred to as the clustering problem, which is an extensively researched problem in computer science. However, the unique requirements of the problem of vehicle–sharing mean that general clustering techniques have limited applicability.

Vehicle–sharing as a form of CT has been considered in industrial and commercial settings. For example, most taxi companies in larger cities have been offering the possibility of shared transportation between a limited number of frequent origins and destinations. Scientifically very little is known about the computational aspects of these vehicle–share operations. However, the computer systems supporting such operations are likely to be semi–automatic, to perform batch–grouping of requests, and to suffer from scalability problems. In comparison, the trip grouping algorithm proposed in this paper is automatic, performs online–grouping or requests, and is highly scalable.

More automatic systems that perform online optimization of vehicle–sharing also exist [3, 15, 22]. These systems however perform a computationally easier task. They either match pairs of trip requests only [15] or are offered from/between a limited set of locations [3, 15, 22]. Additionally, the high volume scalability of these systems has not been demonstrated. Nonetheless, the analysis in [21] and the existence of these systems are evidence that the problem considered by the paper is real and has industrial applications.

Parallel processing of high–volume data streams has been considered by several papers [4, 7, 17, 18, 23, 24, 25]. Some of these study the parallelization of continuous stream queries [7, 23, 24]. GSDM [7] decomposes the computation of a single continuous stream query into a partition, a compute, and a combine phase. In GSDM, the distributed execution strategies are expressed as *data flow distribution templates*, and queries implementing the three phases are specified in separate scripts. In contrast, SCSQ [24] exposes the parallelization phases to the query language so that the distribution patterns becomes part of a single parallel, continuous stream query. This paper utilizes the stream processing engine and query language in SCSQ to express and evaluate different (parallel) stream processing strategies for an RSS.

In GSDM, two different stream partitioning strategies are considered: *window distribute* (WD) and *window split* (WS). In WD, entire logical windows are distributed among compute nodes. In WS, an operator dependent stream split function splits logical windows into smaller ones and assigns them to particular compute nodes for processing. WS has several advantages over WD. First, in applications where the execution time of the stream query scales superlinearly with the size of the logical window, WS provides superior parallel execution performance over WD. Second, in real-time response systems, where the query scales superlinearly, WD is not applicable as it can introduce severe delays in the result stream. Third, in systems where the quality of the results that are computed in parallel are highly dependent on the tuples inside the logical windows of the compute nodes, WD provides inferior results in quality over WS, because individual tuples are not considered in the partition phase. As all of the above three conditions hold in the case of vehicle–sharing, WD is clearly not of interest. WS is similar to the spatial stream partitioning methods presented in this paper in the sense that both presented partitioning methods consider individual tuples in the partitioning process. However, in the static cases no windows are formed over the stream, but rather tuples are assigned to compute nodes based on a general partitioning table. In contrast, in the adaptive cases

windows are formed over the stream, the partitioning table is periodically updated based on the tuples in a window, and then tuples are assigned to compute nodes the same way as in the static case.

Database indices support the efficient management and retrieval of data in large databases. In particular, spatial indices support efficient retrieval of spatial objects, i.e., objects that have physical properties such as location and extent. Spatial indices can be divided into two types: *data partitioning* and *space partitioning* spatial indices [19]. The partitioning mechanisms used in spatial indices have a close relation to the partitioning performed in the present paper.

Data partitioning indices usually decompose the space based on Minimum Bounding Rectangles (MBRs). A primary example is the R–tree that splits space with hierarchically nested, and possibly overlapping Minimum Bounding Rectangles (MBRs) [13]. However, for the application at hand, data partitioning schemes are not well suited for several reasons. They often use a non-disjoint decomposition of space. Consequently, a naïve partitioning based on MBRs could either assign requests to several partitions, and hence later to several shares, or could assign requests from a region where several MBRs overlap to several partitions, thereby potentially eliminating the chance for good matches. While a disjoint partitioning of space could be derived based on the MBRs, computation to derive such a partitioning would be complex and potentially expensive, and the derived partitions will most likely not be balanced.

On the other hand, space partitioning indices decompose the entire space into disjoint cells. These disjoint cells can be based on a regular grid, or on an adaptive grid. Regular grids can result in empty partitions because of skewed data distributions. Hence, a regular grid is not well–suited for the application at hand as it does not support load–balancing.

Quad–trees partition the space into four quadrants in a recursive fashion [6]. Quad–trees divide each region into four equally sized regions, while *point quad trees* [19] allow the size of the regions to be dynamic. Quad–trees have been extended to higher dimensions also. One of the space partitioning methods used in this paper is quite similar to a 1–level deep, four dimensional point quad tree with the exception that in the herein considered space partitioning method a split point is not necessarily a data point. The k–d–tree is a space partitioning spatial index that hierarchically divides each dimension into two along each of the k dimensions [1, 2]. The other partitioning method used in this paper corresponds to a 1–level deep, four dimensional k–d–tree.

3. VEHICLE–SHARING

Large–scale, personalized, on–demand CT systems need efficient and effective computer support. Systems providing this support have two aspects. The first aspect is operational aspect as to how information is communicated between the user and the service provided by the system, and how trip requests are processed. There second aspect is computational or algorithmic and deals with how the optimization of CT is performed. The following subsections study an existing CT system and propose a new one. Section 3.1 formalizes the vehicle–sharing problem, adopted from [12]. Section 3.2 describes the operational aspects of a Cab–Sharing System (CSS)–an instance of a CT system in which the shared vehicles are cabs. Section 3.3 describes the computational or algorithmic aspects of the trip grouping algorithm employed in the CSS. Section 3.4 describes the problems that arise when the trip grouping algorithm is applied in larger scale CT systems. Section 3.5 proposes a Ride–Sharing Service (RSS) and describes its operational requirements. Finally, Section 3.6 describes how the trip grouping algorithm in Section 3.3 can be modified to meet these requirements.

3.1 The Vehicle–Sharing Problem

Let $\mathbb{R}^2$ denote the 2-dimensional Euclidean space, and let $\mathbb{T} \equiv \mathbb{N}^+$ denote the totally ordered time domain. Let $R = \{r_1, \ldots, r_n\}$ be a set of *trip requests* $r_i = \langle t_r, l_o, l_d, t_e \rangle$, where $t_r \in \mathbb{T}$ is the *request time*, $l_o \in \mathbb{R}^2$ and $l_d \in \mathbb{R}^2$ are the *origin* and *destination locations*, and $t_e \geq t_r \in \mathbb{T}$ is the *expiration time*, i.e., the latest time by which the trip request must be accommodated. A trip request $r_i =< t_r, l_o, l_d, t_e >$ is *valid* at time t if $t_r \leq t \leq t_e$. $\Delta t = t_e - t_r$ is called the *wait time* of the trip request. A *vehicle–share* $s \subseteq R$ is a subset of the trip requests. A vehicle–share is valid at time t if all trip requests in s are valid at time t. Let $|s|$ denote the number of trip requests in the vehicle–share. Let $d(l_1, l_2)$ be a distance measure between two locations l_1 and l_2. Let $m(s, d(.,.))$ be a method that constructs a valid and optimal pick-up and drop-off sequence of requests for a vehicle–share s and assigns a unique distance to this sequence based on $d(.,.)$. Let the *savings* p for a trip request $r_i \in s$ be $p(r_i, s) = 1 - \frac{m(s,d(.,.))/|s|}{m(\{r_i\},d(.,.))}$. Then, the *vehicle–sharing problem* is defined as follows.

DEFINITION 1. *For a given* ***maximum vehicle–share size*** K, *and* ***minimum savings*** min_savings $\in [0, 1]$, *the vehicle–sharing problem is to find a disjoint partitioning* $S = \{s_1 \uplus s_2 \uplus \ldots\}$ *of* R, *such that* $\forall s_j \in S$, s_j *is* valid, $|s_j| \leq K$, *and the expression* $\sum_{s_j \in S} \sum_{r_i \in s_j} p(r_i, s_j)$ *is maximized under the condition that* $\forall r_i \in s_j \; p(r_i, s_j) \geq$ *min_savings* or $\{r_i\} = s_j$.

3.2 Overview of the Cab–Sharing System

The Cab–Sharing System (CSS) proposed in [12] is a Location–Based Service (LBS) in the transportation domain. In its most simple form, it is accessible to the user via a mobile phone through an SMS interface. The components and operation of the CSS is depicted in Figure 1 and can be described as follows. The user inputs two addresses with an optional maximum time that s/he is willing to wait. The service in turn then:

1. geocodes the addresses,
2. calculates an upper bound on the cost of the fare,
3. validates the user's account for sufficient funds,
4. submits the geocoded request to a pool of pending requests,
5. within the maximum wait time period finds a nearly optimal set of "closeby" requests using a number of heuristics (described in Section 3.3),
6. delivers the information about the set (request end points, and suggested pickup order) to the back–end cab dispatch system,

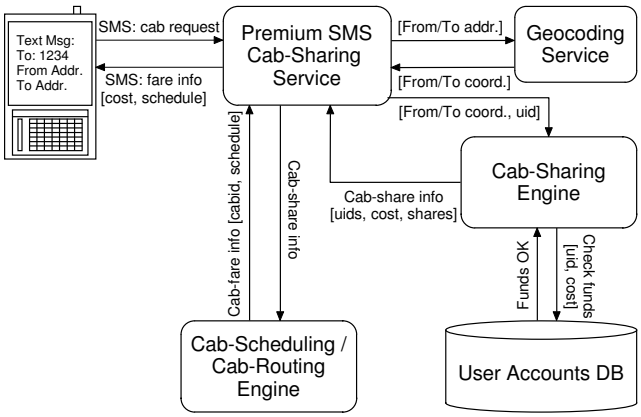


Figure 1: Cab–sharing service components and process.

7. delivers information about the fare (estimated time or arrival, cost, savings, etc...) to the involved users.

3.3 A Trip Grouping Algorithm

Finding the optimal solution to the vehicle–sharing problem is computationally difficult. Given n requests, the number of possible disjoint partitionings, where the size of the vehicle–shares is exactly K is:

$$\binom{n}{K} \times \binom{n-K}{K} \times \cdots \times \binom{2K}{k} \times \binom{k}{k} = \frac{n!}{\lceil n/K \rceil \times K!}.$$

In the case of $n = 100$ and $K = 4$, this expression evaluates to a number that has 155 digits. The number of possible disjoint partitions, where the size of the vehicle–shares is at most $K = 4$ is even larger. Clearly, evaluating all possible options and selecting the most optimal one is not a feasible approach. Instead, the Trip Grouping (TG) algorithm at the heart of the CSS tries to derive a nearly optimal solution by employing a number of heuristics and approximations. The steps of the TG algorithm along with the applied heuristics and approximations are described next.

1. Distinguish between the set of expiring trip requests (R_x) and all valid requests (R_q). Wait with mandatory grouping of trip requests until expiration time. A request can also be grouped into a vehicle–share before its expiration time with another expiring request. This *lazy* heuristic does not make the algorithm miss out on an early cost–effective grouping for the request, but rather gives the requests more opportunities to be part of a grouping.
2. Based on the distance measure $d(.,.)$, define a pair–wise *fractional extra cost* (FEC) between two requests and calculate it for every pair of expiring and valid requests. In the TG algorithm the fractional extra cost between two requests r_i and r_j (w.r.t. r_i) is defined as $FEC(r_i, r_j) = \frac{d(r_i.l_o, r_j.l_o) + d(r_i.l_d, r_j.l_d)}{d(r_i.l_o, r_i.l_d)}$. In the case when the distance measure $d(.,.)$ is the Euclidean distance, the calculations of fractional extra costs between three requests r_1, r_2, and r_3 (w.r.t. r_1) are shown in Figure 2. Note that the defined fractional extra cost is an upper bound on the true fractional extra cost, as there may be a shorter route than to serve the requests in the order assumed by the fractional extra cost calculation, i.e., $r_j.l_o \rightarrow r_i.l_o \rightarrow r_i.l_d \rightarrow r_j.l_d$.
3. Consider the best, i.e., lowest cost / highest savings, K–sized vehicle–share for an expiring request $r_i \in R_x$ to be composed of the first K requests with lowest FEC for r_i. This heuristic assumes that pair–wise fractional extra costs are additive.
4. Estimate the Amortized Cost (AC) of a vehicle–share s (w.r.t. r_i) as the normalized cumulative sum of FECs as: $AC(r_i, s) = \frac{1 + \sum_{r_j \in s} FEC(r_i, r_j)}{|s|}$. This heuristic assumes that there exists an optimal pick–up and drop–off sequence for requests in s, such that the cost of this sequence $m(s, d(.,.)) \leq AC(r_i, s) * d(r_i.l_o, r_i.l_d)$.
5. Greedily group the best maximum K-sized vehicle–share that has the minimum amortized cost over all expiring trip requests. This heuristic is greedy because it possibly assigns a not–yet–expiring request r_j to a vehicle share of an expiring request, without considering what the current or even future best vehicle–share would be for r_j.
6. Remove requests that are part of the best vehicle–share from further consideration.
7. Repeat steps 2 through 7 as long as the best vehicle–share meets the minimum savings requirement.
8. Assign remaining trip requests to their own (single person) "vehicle–shares".

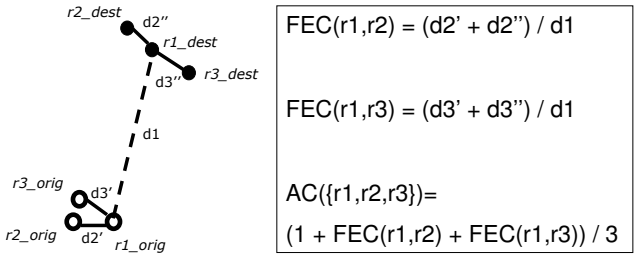


Figure 2: Illustration of fractional extra cost (FEC) and amortized cost (AC) calculations w.r.t. request r1.

Even though the TG algorithm is based on heuristics, estimations and assumptions, in [12], it has been found to effectively optimize the vehicle–sharing problem. Furthermore, while some assumptions about extra costs for vehicle–shares do not hold in all cases, the combination of the approximations and assumptions result in an estimated cost for the vehicle–shares that is higher than the true minimum cost if the optimal pick–up and drop–off sequence is considered.

3.4 Problems with Large–Scale CT Systems

Unfortunately, the TG algorithm cannot be naïvely applied to facilitate a large–scale CT system, such as a ride–sharing system. The TG algorithm needs to calculate the pairwise fractional extra costs between expiring requests and all requests in the queue, entailing on the order of $O(n^2)$ cost calculations. In [12] a simple but effective implementation of the TG algorithm was able to handle loads of up to 50,000 requests per day, during which at peak traffic hours the num-

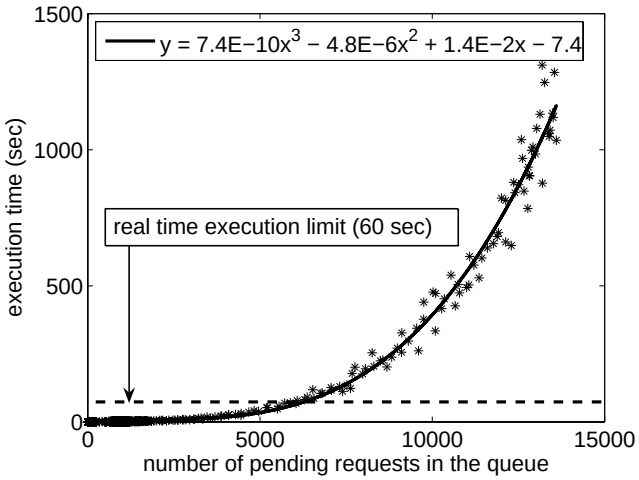


Figure 3: Scalability problems of the general trip grouping algorithm.

ber of requests within 10 minutes was at most 2,500. However, as input sizes increase, the execution times of any serial implementation of the TG algorithm will reach a point where continuous grouping is not possible. Then, the algorithm is not able to find nearly optimal groups for all the expiring request before they actually expire. This is demonstrated in Figure 3, where a load of 250,000 requests with common wait times of 10 minutes are grouped minute–by–minute using a highly efficient implementation of the TG algorithm. This implementation of the TG algorithm is able to keep up with the request flow most of the time, but when the number of pending requests exceeds about 5,200 (during rush hour), it is not able to find groups for the expiring requests within the allowed execution time of 60 seconds. In the example the *grouping cycle time* of the TG algorithm is 60 seconds, i.e. the algorithm is responsible for grouping the request that will expire within the next 60 seconds. Altering this grouping cycle time does not eliminate the problems of the algorithm in the case of large input sizes. Figure 3 also reveals that the computational complexity of the implementation of the TG algorithm is $O(n^3)$. This is due to the fact that, as described by the third heuristic in Section 3.3, the best K–sized vehicle–share is composed of the first K requests with lowest FEC for an expiring request. This necessitates a linear–time top–K selection for each expiring request, making the algorithmic complexity of the TG algorithm $O(n^3)$. Consequently, the above described scalability problems severely limit the applicability of the TG algorithm in a large–scale CT system.

3.5 Ride–Sharing Application Requirements

Ride–sharing is a type of vehicle–sharing where private vehicles are used as transportation. This fact represents additional requirements on solution to the general trip–sharing problem. In the context of ride–sharing there are *ride–requests* and *ride–offers*. Ride–requests are synonymous to trip requests both in form and semantics, with the exception that ride–requests do not necessarily have to be served. Ride–offers have at least three important attributes in addition to the attributes of a trip request. The first attribute specifies whether the offering person is willing to leave his/her vehicle behind. A person offering a ride with willingness of leaving his/her vehicle behind is either willing to take alternate modes of transportation or relies on the efficient operation of the ride–sharing system for future trips until he/she returns to his/her vehicle. A person not willing to leave his her vehicle behind values or needs his/her independence throughout the day. The second attribute specifies a *maximum relative extra cost* the offering person is prepared to incur. Finally, the third attribute specifies the *maximum number of additional passengers* the offering person's vehicle can accommodate.

3.6 Application of the TG Algorithm in a RSS

It is clear that the TG algorithm cannot be applied in its current form for a ride–sharing application. However, a few simple modifications can make it applicable. First, in the context of ride–sharing, the ride offering person would like to leave as soon as the best vehicle–share that can be constructed meets the maximum relative extra cost requirements of the ride–offer. Hence, it makes sense to prioritize the order of greedy grouping based on the time the ride–offers have been present in the system. Second, because maximum relative extra cost requirements are defined by ride–offers individually, in every grouping cycle (execution of the TG algorithm) the best vehicle–share for *all* ride–offers needs to be considered. Third, every vehicle–share needs to fulfill the following two conditions: 1) it can contain only one ride–offer where the offering person is not willing to leave his/her vehicle behind, and 2) it has to contain at least one ride–offer of any type. To fulfill the above conditions it is enough to distinguish between two different sets: 1) the set of ride–offers of either type $\{R_o^{\bar{o}} \cup R_o^o\}$, and 2) the joint set of ride–request and ride–offers where the offering person is willing to leave his / her vehicle behind $\{R_r \cup R_o^o\}$. Associating these sets to sets used by the TG algorithm as $R_x = \{R_o^{\bar{o}} \cup R_o^o\}$ and $R_q = \{R_r \cup R_o^o\}$, the vehicle shares constructed by the TG algorithm fulfill the above two conditions.

Obviously, the modifications to the TG algorithm that are necessary to facilitate the proposed RSS are straight–forward. However, to preserve clarity in representation, the remainder of the paper considers only the implementation of a highly scalable TG algorithm.

4. HIGHLY SCALABLE TRIP GROUPING

Although the TG algorithm can be modified to meet the unique requirements of the proposed RSS, as it was demonstrated in Section 3.4, the algorithm in its present form does not scale with the input size and hence cannot be applied in large scale CT systems, such as the proposed RSS. This section describes a parallel implementation of the TG algorithm in the SCSQ Data Stream Management System.

Queries and procedures in SCSQ [23] (pronounced *sis–queue*) are specified in the query language SCSQL [24] (pronounced *sis-kel*). SCSQL is similar to SQL, but is extended with streams as first-class objects. SCSQ also features a main memory database. This database is used to keep the trip requests that are waiting, along with statistics about the data distributions. The waiting requests are processed by the TG algorithm and the statistics are used by the partitioners.

Details of the implementations are organized as follows. Section 4.1 describes how the trip grouping algorithm is implemented as a stored procedure in SCSQL. Section 4.2 outlines how SCSQ allows parallelization of the continuous stream query implementation of the TG algorithm. Section 4.3 describes four spatial partitioning methods that are used to partition the stream of trip requests into sub-streams for parallelization purposes.

4.1 Processing of a Request Stream

The TG algorithm is expressed as a procedure in SCSQL, which is listed below.

```
(1) create function tg(vector input_window,
(2)                    integer K, real min_savings,
(3)                    integer wait_time)->vector
(4) as begin
(5)   declare vector ex, vector bcss, timeval ct;
(6)   insert_q(in(input_window));
(7)   set ct = get_end(input_window);
(8)   set ex = select_ex_q(curr_time, wait_time);
(9)   set bcss = {};
(10)  for each vector r where r = in(ex)
(11)  begin
(12)     remove_q(r);
(13)     set s = select subvector(ac,0,i)
(14)             from  vector fec, vector ac,
(15)                   integer i, integer k
(16)             where fec = topk(calc_FEC(r),2,K)
(17)               and ac = calc_AC(fec,2)
(18)               and i = min(ac,2);
(19)     if savings(s) >= min_savings
(20)     begin
(21)        set bcss = concat(bcss,members(s));
(22)        remove_q(members(s));
(23)     end;
(24)     else
(25)        set bcss = concat(bcss,r);
(26)  end;
(27)  result bcss;
(28)end;
```

The `tg` procedure takes an `input_window` of the most recently arrived trip requests, and the three algorithm parameters `K`, `min_savings`, and `wait_time` The output of `tg` is a vector of best vehicle–shares, `bcss`. `tg` executes as follows. First, on line 6, all requests in `input_window` are added to the main memory table of waiting requests `q`. Then, on line 7, based on the `wait_time` parameter and the current time `ct` (indicated by the end of the `input_window`), expiring requests, `ex`, are selected from `q`. The `for each` loop on line 10 iterates over each request `r` in `ex` as follows. On line 12, the request `r` is removed from the `q`. Then, in a compound query on lines 13–18, the best, maximum `K`–sized vehicle–share for `r` is found. The first part of the compound query, on line 16, calculates the fractional extra costs `calc_FEC(r)=<r,ri,fec>` between `r` and all other requests in `q`, and selects the tuples for the `K` requests with the lowest fractional extra costs. The remaining parts of the compound query, on lines 17–18, calculates the amortized costs `calc_AC(fec)=<r,ri,ac>` based on the top–K fractional extra costs, and selects the lowest of these costs. The best vehicle–share that corresponds to this lowest amortized cost is assigned to `s` on line 13. Finally, if the `savings` of `s` is greater than equal to `min_savings`, then the `members` of `s` are added to the the best vehicle–shares, `bcss` (line 21), and are removed from `q` (line 22). Otherwise, `r` could not share its trip, and will be the only one in its vehicle–share (line 25). The implementations of the derived functions `insert_q`, `get_end`, `select_ex_q`, `remove_q`, `subvector`, `calc_FEC`, `savings`, and `members` are omitted to preserve brevity. For efficiency reasons, core functions that need to iterate over a set, such as `topk` and `calc_AC` are implemented as foreign functions in Lisp. Foreign functions allow subroutines defined in C/C++, Lisp, or Java to be called from SCSQL queries. The implementation of these functions is also omitted.

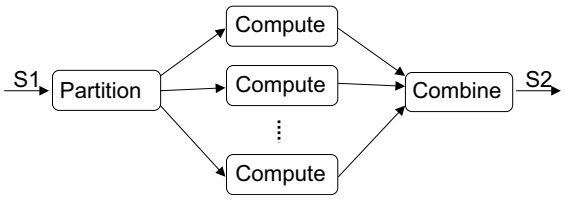


Figure 4: Communication pattern of TGs working in parallel.

4.2 Parallel Stream Processing in SCSQ

Apart from streams, SCSQL includes Stream Processes (SPs) as first–class objects in queries. SPs allows dynamic parallelization of continuous queries, which is used in this paper to divide the incoming trip requests. The user associates subqueries with SPs. Massively parallel computations are defined in terms of sets of parallel subqueries, executing on sets of SPs.

The output of an SP is sent to one or more other SPs, which are called *subscribers* of that SP. The user can control which tuples are sent to which subscriber using a *postfilter*. The postfilter is expressed in SCSQL, and can be any function that operates on the output stream of its SP. For each output tuple from the SP, the postfilter is called once per subscriber. Hence, the postfilter can transform and filter the output of an SP to determine whether a tuple should be sent to a subscriber. Postfilters are used in the experiments to partition the input stream between the SPs that are carrying out TG.

The divide–and–conquer experiments are expressed as queries in SCSQL. All these queries have the same communication pattern between SPs, as shown in Figure 4. A Partition SP reads a stream of incoming trip requests (S1). That stream is partitioned into partial streams, which are sent to the Compute SPs. Each Compute SP executes the `tg` procedure on its partial stream. Also, each Compute SP evaluates the savings achieved, by comparing the total cost of all trips with the total cost of the shared trips. The results of all Compute SPs are merged together by a Combine SP. The resulting stream of cab requests (S2) is sent to the user.

4.3 Spatial Partitioning Methods

Section 3.4 showed that the TG algorithm does not scale well enough for large-scale CT systems. The key idea to overcome the scaling issue is a divide–and–conquer approach. Each request $r_i = \langle t_r, l_o, l_d, t_e \rangle$ are characterized by its origin and destination locations, $l_o \in \mathbb{R}^2$ and $l_d \in \mathbb{R}^2$. Hence,

a request can be geographically characterized by a point in $l_o \times l_d$. In other words, a request is characterized by a point in $\mathbb{R}^4$. The divide–and–conquer approach is to partition this space and assign each partition to one TG. Intuitively, this approach will gain in execution time since each TG algorithm has less workload, but will lose some of the vehicle–sharing opportunities since none of the partitions are able to probe all combinations that a serial implementation can do. The goal is to find a partitioner that executes efficiently and achieves maximum savings. The following partitioning strategies are implemented in SCSQL and investigated experimentally.

4.3.1 Baseline Queries

Two baseline queries are executed; the unpartitioned query and the round–robin query. These queries form a performance baseline of the best and worst possible savings and execution speeds. All other methods should be compared to the measurements of these two queries.

The *unpartitioned* query applies a single TG algorithm on the entire request stream without any partitioning. Since all requests are going to a single TG, all possible sharing opportunities will be investigated. The unpartitioned query will give the best savings, but it will also take the longest time to execute because all burden will be placed on a single node. The unpartitioned query is expressed in SCSQL as follows:

```
select tg(v, 4, 0.8, 600, 60)
from vector v, charstring file
where v = twinagg(streamfile(file), 60.0, 60.0)
and file in
{"L16.dat","L8.dat","L4.dat","L2.dat","L1.dat"};
```

The `streamfile(file)` function reads tuples that are stored in `file`, and streams them out. The `twinagg(inputstream, size, stride)` function is taking a stream as the first argument and emits a time window over the last `size` seconds, every `stride` seconds. Hence, if `size=stride`, `twinagg` emits tumbling (consecutive and non–overlapping) windows of the input stream. This `twinagg()` makes sure that `tg()` always will get one minute worth of requests each time. Hence, `tg()` will get called once per minute. If no requests have arrived during a certain minute, `twinagg()` will emit an empty window for that minute. `tg(input_window, K, min_savings, wait_time)` performs the trip grouping algorithm. The query is executing once per file in the collection of filenames given on the last line of the query.

The *round–robin* partitioner will send the first request to one working SP. The next request will be sent to another working SP, and so on. This way, each SP will be given exactly $1/n$ of the total load, so the load balance is perfect. Since the round–robin partitioning scheme is perfectly load balanced, it will achieve the maximum possible execution speed. On the other hand, a TG algorithm executing in an SP that is operating on a round–robin data partition can be expected to give inferior savings since nearby requests not necessarily go to the same TG. Thus, the round–robin partitioner is expected to achieve the least savings. It is expressed in SCSQL as:

```
select merge(b)
from bag of sp b, sp c, integer n, charstring file
where b = spv(select streamof(tg(twinagg(stract(c),
              60.0, 60.0), 4, 0.8, 600)))
              from integer i where i=iota(1,n))
and   c = sp(winagg(streamfile(file),n,n),n,'rr')
and n in {16,8,4,2}
and file in
{"L16.dat","L8.dat","L4.dat","L2.dat","L1.dat"};
```

In this query, the output of `streamfile` is passed into `winagg(input_ stream, size, stride)`, which is forming tumbling windows of size `n`, `n` being the number of subscribers to the Partition SP `c`. Each window is an ordered set of tuples, so it is represented as a vector. The round–robin function `rr`, is applied once per subscriber. For subscriber i, `rr` picks up the i-th element in the vector emitted from winagg. The `SP(stream, nsubscribers, postfilter)` is assigning `stream` and `postfilter` to a new SP, which should expect `n` subscribers. Thus, a combination of a `winagg` on a stream and a vector dereference in the postfilter function results in a round–robin partitioner.

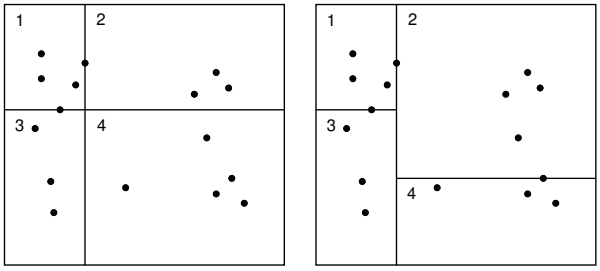
`iota(m,n)` generates all integers from m to n. Hence, the query in the call to `spv(bag of stream)` creates `n` duplicates of the query `streamof(tg(twinagg(stract(c),60.0, 60.0), 4, 0.8, 600))`, where `stract(c)` is extracting the stream from stream process `c`. Each one of these queries will be assigned to a stream process. Finally, the output of all the stream processes in b will be merged. Refer to Figure 4 for a graphical representation of the communication pattern: The partition is done at SP `c`, compute is performed by the SPs in `b`, and the combination is done in the `merge` at top level.

4.3.2 Static Point Quad Partitioning

Static point quad partitioning (SPQ) calculates from historical data the medians of each dimension of the trip requests. Each dimension of the four–dimensional trip request data space split once along the median of each dimension. Figure 5(a) shows the SPQ partitions for some data points in two dimensions. By splitting each dimension once, SPQ partitions the four–dimensional trip request data space into 16 regions. One or more regions can be assigned to one SP, executing a TG algorithm for that region. This SCSQL query executes SPQ:

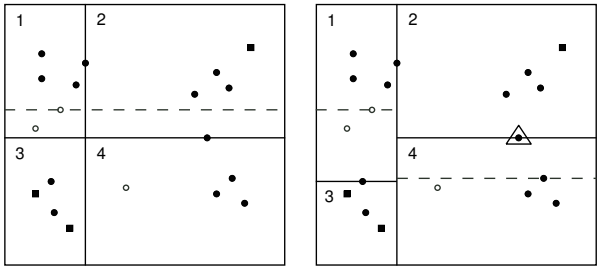
```
select merge(b)
from bag of sp b, sp c, integer n, charstring file
where b = spv(select streamof(tg(twinagg(stract(c),
              60.0, 60.0), 4, 0.8, 600)))
              from integer i where i=iota(1,n))
and   c = sp(streamfile(file),n,'pq')
and n in {16,8,4,2}
and file in
{"L16.dat","L8.dat","L4.dat","L2.dat","L1.dat"};
```

The difference between this query and the round–robin query above is only in the call to the partitioning SP `c`. Instead of applying postfilter function `rr` on a window, SP `c` is applying the `pq` postfilter on the tuples from `streamfile`. For each tuple, `pq` decides which subscriber it should go to.



(a) SPQ (b) SKD

Figure 5: Illustrations of the static partitioning methods.



(a) APQ (b) AKD

Figure 6: Illustrations of the dynamic partitioning methods.

4.3.3 Static KD Partitioning

Static KD partitioning (SKD) splits trip request data in a hierarchical fashion by processing dimensions one after the other as follows. For a given dimension, SKD first calculates the *local* median for that dimension, and then splits the local trip request data for the dimension based on the median into approximately equal sized subsets. Figure 5(b) shows the SKD partitions for some data points in two dimensions. The data is first split around the median of the horizontal dimension, then the data in each of the so obtained partitions is further split around the local (horizontal) median of each of the partitions. By splitting once per dimension, the KD also partitions the four–dimensional trip request data space into 16 regions. The SCSQL query that executes SKD differs from that of SPQ in that it applies another postfilter function at the partitioning SP, namely `kd` instead of `pq`. Since the difference is so small, the SCSQL query is not shown here.

4.3.4 Adaptive Point Quad Partitioning

The trip request data distribution changes over time. During the morning rush hours people want to move from their homes (residential district) to their work (business and industrial districts). During the evening rush hours the opposite is true. The trip requests that correspond to the morning rush hour movements are likely to fall in different partitions than the trip requests that correspond to the evening rush hour movements. Consequently, the "morning rush hour" partitions will be densely populated in the morning hours, and the "evening rush hour" partitions will be densely populated in the evening hours. Clearly, a static partitioning method does not consider these temporal changes in data distribution and is therefore likely to result in temporarily unbalanced partitions.

The adaptive point quad partitioning (APQ) adjusts the boundaries of the partitions periodically, based on statistics obtained from a recent history buffer of the trip request stream, and distributes the newly arriving trip requests according the newly adjusted partitions. Figure 6(a) shows two consecutive partitionings that are constructed by the APQ partitioning for some data points in two dimensions. Hollow dots represent data points that were present when the previous partitioning was constructed, but are not present or are not relevant for the construction of the current partitioning. In contrast, solid rectangular markers represent data points that were not present when the previous partitioning was constructed, but are relevant for the construction of the current partitioning. Solid and dashed lines represent current and previous partition boundaries. The following SCSQL query executes TG algorithm with APQ:

```
select merge(b)
from bag of sp b, sp c, integer n, charstring file
where b = spv(select streamof(tg(twinagg(stract(c),
              60.0, 60.0), 4, 0.8, 600)))
              from integer i where i=iota(1,n))
and   c = sp(pqstat(streamfile(file),
             600.0, 60.0, 10),n,'pq')
and n in {16,8,4,2}
and file in
{"L16.dat","L8.dat","L4.dat","L2.dat","L1.dat"};
```

This query differs from the SPQ query in the call to the partitioning SP `c`. The `streamfile` function is wrapped by `pqstat(inputstream, size, stride, samplefreq)`. This function emits the same stream as its input stream, and maintains statistics in a main memory table of SCSQ. Every `stride`×`samplefreq` seconds, `pqstat` computes medians in each dimension of $l_o \times l_d$ across the tuples seen in the last `size` seconds. These median values are then used in the `pq` postfilter. This way, the partitioning decisions are always done on recent data.

4.3.5 Adaptive KD Partitioning

The adaptive KD partitioning (AKD) adjusts the boundaries of the partitions periodically, based on statistics obtained from a recent history buffer of the trip request stream. and distributes the newly arriving trip requests according the newly adjusted partitions. Figure 6(b) shows two consecutive partitionings that are constructed by the AKD partitioning for some data points in two dimensions. The semantics of the symbols used in the figure are the same as in the case of the APQ partitioning. However, Figure 6(b) depicts a situation that can happen in either one of the adaptive spatial partitioning methods. Consider the data point inside the triangle. Since it was present when the previous partition was constructed it has been assigned to compute node 2 for processing. According to the newly constructed partitions however, it should be assigned to compute node 4. To avoid communication between compute nodes, the following design choice is made: once a data point is assigned

to a partition (compute node), it is never reassigned to another partition, even if the newly adjusted partitions would suggest this.

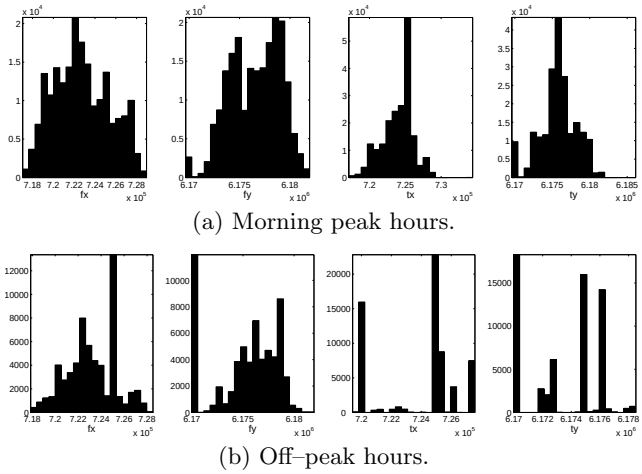
The SCSQL query that executes SKD differs from SPQ in that it applies another statistics wrapper function and another postfilter function at the partitioning SP, namely `kdstat` instead of `pqstat` and `kd` instead of `pq`. `kdstat` works analogously to `pqstat` with the difference that it maintains dynamical versions of local dimension splits of the kind that SKD has. Since the difference between this query and the APQ query is so small, the SCSQL AKD query is not shown here.

5. DENSITY–BASED SPATIAL STREAM PARTITIONING

In all spatial partitioning methods proposed in this paper, the space of requests is split by planes. The locations of the splitting planes is determined by the medians of request data. These splitting planes potentially eliminate the discovery of good shares, when members of the good shares are on different sides of a splitting plane. This naturally leads to some degradation in the overall grouping. The degradation is larger when the planes are cutting through denser regions of the request space with many sharing opportunities, than when the planes are cutting through sparser regions of the request space. Since neither of the proposed partitioning methods consider the distribution density of the requests, the degradation of grouping quality due to boundary effects can be expected to be approximately the same for all four partitioning methods. However, as Section 6 demonstrates, this degradation is rather small.

No matter how small the degradation is, simple spatial partitioning methods that take into account the density of the data could reduce the degradation. The objective of such a density–based partitioning is to determine the positions of the splitting planes so that they pass through regions where data is sparse. To achieve this, a simple but effective clustering method [9] can be used to find local minima in the multimodal data distributions along each dimension, and place splitting planes at those locations. Figure 7 shows the distributions for each dimension of the request data during morning peak hours and off–peak hours. During the morning peak hours, there does not seem to be any regions where the request data is very sparse. However, during off–peak hours, when people who are not working are most likely to be in one of the larger shopping malls, the distributions of the destination dimensions (tx, ty) are clearly multimodal. In this later situation, ensuring that splitting planes are chosen correctly at local minima would minimize the boundary effects. However, since most of the requests are during peak hours, the overall average grouping achieved by the parallel TG algorithm would not be substantially improved.

Since the local minima are likely not to be at the median values of the dimensions, there exists a trade–off between equal–sized partitions and partitions with minimal boundary effects. A dual–objective partitioning that takes this trade–off into consideration could weigh the expected degradation against the imbalance between the created partitions. Although the implementation of the density–based and the dual–objective spatial stream partitioning methods is straight–forward, it is left for future research.



(a) Morning peak hours.

(b) Off–peak hours.

Figure 7: Request data distributions along each dimension. "f" and "t" stand for "from" and "to", respectively. Hence, fx and fy are request origin dimensions, while tx and ty are request destination dimensions.

The proposed spatial stream partitioning methods are devised to scale the TG algorithm to very large flows of requests. However, they can be considered as a general approach to make computationally intensive spatial analysis tasks scalable through parallelization. For example, the density–based and the dual–objective spatial stream partitioning methods can be applied to speed up spatial clustering of streams, spatio–temporal rule mining [10], or the processing of high–resolution image streams.

6. EXPERIMENTS

The parallel implementations of the TG algorithm were tested on a cluster of Intel® Pentium® 4 CPU 2.80GHz PCs. Each SP in the query language started a running process (RP) on a separate node in the cluster. TCP/IP over Fast Ethernet was used to carry streams between the nodes.

Trip request data was simulated using ST-ACTS, a spatio–temporal activity simulator [11]. Based on a number of real world data sources, ST–ACTS simulates realistic trips of approximately 600,000 individuals in the city of Copenhagen, Denmark. For the course of a workday, out of the approximately 1.55 million generated trips, approximately 251,000 trips of at least 3–kilometer length were selected and considered as trip requests. To test the scalability of each of the parallel implementations using the four spatial stream partitioning methods, decreasing sized subsets of the total load of 251,000 trip requests were constructed by only considering every second, fourth, eighth and sixteenth trip request in the input stream. These subsets are referred to as 1/2, 1/4, 1/8, 1/16 load, respectively.

To evaluate the effectiveness of the four spatial stream partitioning methods, for the purposes of parallelization of the TG algorithm, two measures were used: (overall) execution time and average savings achieved by the grouping (also referred to as the quality of the grouping or quality for short).

load	execution time (sec)	savings
0.06125	28.8	0.325
0.125	120.1	0.388
0.25	702.9	0.445
0.5	16343.5	0.491
1	69771.6	0.530

Table 1: Performance of the serial TG algorithm.

The reported savings for each vehicle–share are based on amortized costs, which has been shown to overestimate the true cost of a vehicle–share that considers the optimal pick–up and drop–off sequence of requests. Hence, the reported savings underestimate the true savings. Nonetheless, the reported savings can be used as an unbiased measure for the quality of the grouping.

For each of the partitioning methods an extensive set of experiments were performed for fixed algorithm parameters ($K = 4$, `min_saving` $= 0.2$, and $\Delta t = 10$ minutes) under varying loads using degrees of parallelization. The adaptive partitioning methods updated the partitions every 10 minutes based on the trip request that arrived in the last 10 minutes.

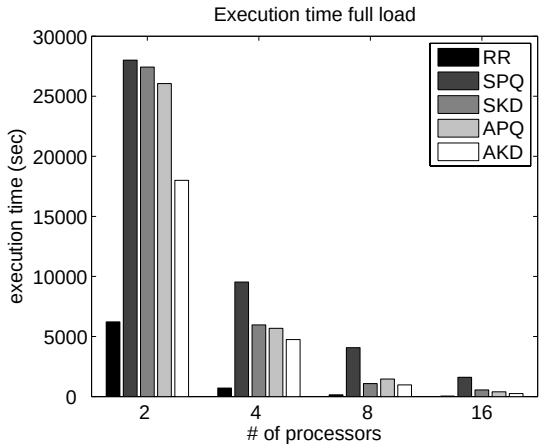
6.1 Baseline Performance

To establish a point of reference for the performance measures the baseline queries specified in Section 4.3.1 were executed. Table 1 shows the results for the unpartitioned query. Savings obtained by the unpartitioned query (serial execution) are *considered* to be optimal, while running times are *considered* to be worst case performance. Note that these measures are "optimal" and "worst case" with respect to the TG algorithm. Moreover, as it is demonstrated in Section 3.3, due to the computational complexity of the *vehicle–sharing problem*, the calculation of a truly optimal grouping, even in the case of a few requests, is infeasible. Due to the large difference in scale between serial and parallel execution times, serial execution times are not shown in subsequent figures. Savings achieved by the unpartitioned query (serial execution) are also not shown in subsequent figures, but are used to report relative performance of the parallel executions in terms of savings and quality.

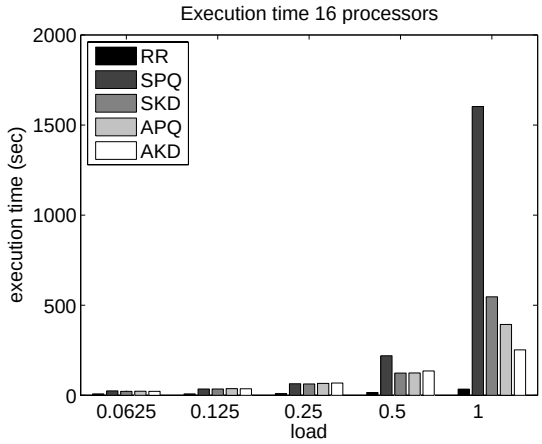
In comparison, the round–robin query was executed to obtain optimal execution times due to perfect load balancing and worst case savings due to the distribution independent partitioning of requests between SPs. The results of these experiments are shown in Figures 8 and 9 as RR, however it is emphasized that RR is *not* one of the proposed spatial stream partitioning methods, but is *only* used as a reference.

6.2 Absolute Performance of the Parallel TG Algorithms

Figures 8 and 9 show the absolute performance of the parallel TG algorithm for varying load and degrees of parallelization using different spatial stream partitioning methods. From Figure 8(a) it can be seen that the execution times of all of the methods decrease as the parallelism is increased. Figure 8(a) also reveals that the adaptive versions of the spatial partitioning methods adjust well to the changing spatial distribution of the requests, resulting in more



(a) Performance for full load.

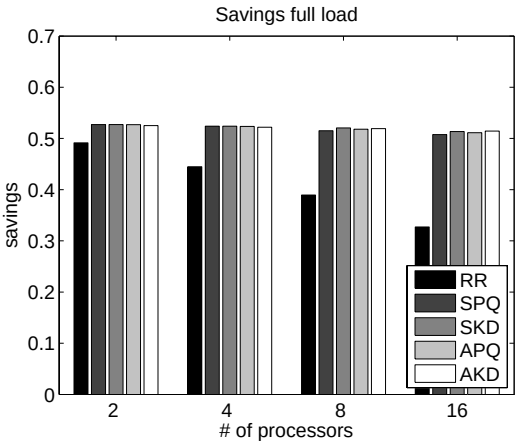


(b) Performance for 16 processors.

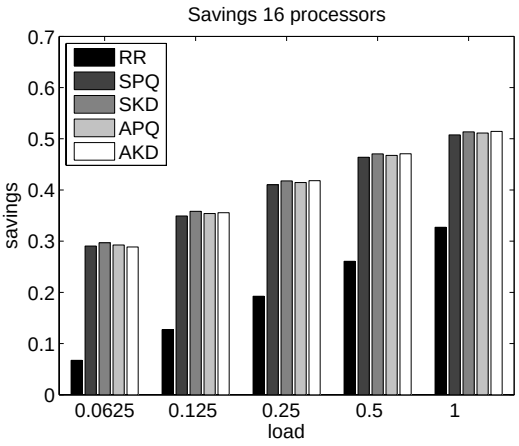
Figure 8: Execution times for the parallel TG algorithm for different partitioning methods for varying parallelization and load.

balanced partitions and ultimately faster execution times when compared to their static version. The improvement in execution time due to adaptive partitioning is most evident for the SPQ partitioning. Figure 8(b) shows that while the execution time of the TG algorithms can be scaled, the underlying algorithmic complexity of the TG algorithm executed on the compute nodes does not change. The effect of the underlying algorithmic complexity is more observable for spatial partitioning methods that construct less balanced partitionings, in particular SPQ.

Figure 9(a) shows that in general the quality of the grouping decreases as the degree of parallelization is increased. However in the case of non–spatial partitioning (RR) this degradation is significant, while in the case of either one of the four spatial partitioning methods it is negligible. Figure 9(b) shows that the grouping quality increases as the load is increased. This is due to the simple fact that the spatio–temporal density of the trip requests increases. As a consequence, the likelihood that a request becomes part of a "good" vehicle share increases. The almost negligible differences between the qualities achieved by the four parti-



(a) Quality for full load.



(b) Quality for 16 processors.

Figure 9: Savings for the parallel TG algorithm for different partitioning methods for varying parallelization and load.

tioning methods, as explained in Section 5, is due to the fact that since neither of the partitioning methods consider the data densities, but only the medians of the dimensions, the total degradation due to boundary effects is approximately the same for the four partitioning methods.

6.3 Relative Performance of the Parallel TG Algorithms

Figure 10 shows relative execution times of the parallel TG algorithms when compared to the optimal execution time that is achieved by RR partitioning due to perfect load balancing. With the exception of the SPQ partitioning, all other partitioning methods result in parallel execution times that are within the same order of magnitude as the optimal. There are potentially two sources for this slowdown: the cost of partitioning and the extended execution times due to improper load balancing. Since adaptive partitioning methods have to maintain a limited history of the stream and periodically recompute partition boundaries based on this history, they do additional work compared to their static counterparts. Figure 10 shows that execution times resulting from adaptive partitioning are significantly shorter than the execution times achieved by static partitioning. Hence, it is clear that the additional time needed to perform the spatially partitioned parallel queries can mainly be attributed to unbalanced partitions.

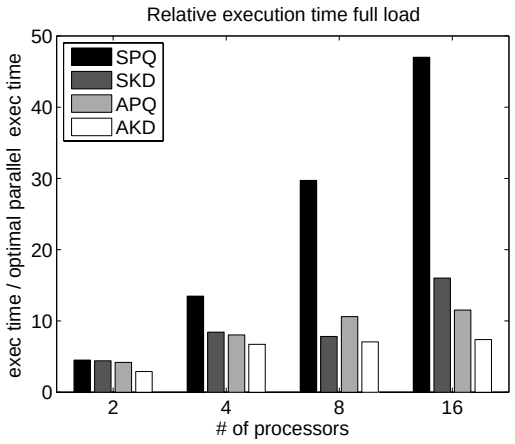


Figure 10: Relative performance for the parallel TG algorithm (compared to RR partitioning) for different partitioning methods for varying parallelization.

Finally, comparing the savings in Figure 9(b) to the savings in Table 1 reveals that the grouping quality achieved by either one of the partitioning methods is within the 95% of the optimal quality for the full load. Even if the load is decreased to 1/16 of the total load, all the spatial partitioning methods still achieve approximately 90% of the maximum possible savings.

The experiments can be summarized as follows. First, RR partitioning has perfect load balance and is a very simple partitioning method, hence it has the fastest execution time. However, RR partitions the space badly and achieves a bad grouping quality. Second, using a spatial partitioning method improves grouping quality. All spatial partitioning methods achieve at least 95% of the maximum possible savings in the case of the full load. Third, the adaptive partitioning methods always execute faster than their static equivalents. That is because the adaptive methods constantly adapt the partitioning according to the last tuples observed, which will lead to better load balance. At the same time, the savings are approximately the same for both the static and adaptive partitionings. Adaptive partitioning is also preferred from an operational point of view, since it does not need any prior knowledge about the data distribution. Finally, since all partitioning methods (except RR) achieve about the same savings, the preferred method is the one with the fastest execution time of SPQ, SKD, APQ, and AKD. Thus, AKD is the best partitioning method.

7. CONCLUSIONS AND FUTURE WORK

This paper proposed highly scalable algorithms for trip grouping to facilitate large–scale collective transportation systems. The algorithms are implemented using a parallel data stream management system, SCSQ. First, the basic trip grouping algorithm is expressed as a continuous stream query to allow for a very large flow of requests. Second, following the divide–and–conquer paradigm, four spatial stream partitioning methods are developed and implemented to divide the input request stream into sub–streams. Third, using the infrastructure of SCSQ and the partitioning methods, parallel implementations of the grouping algorithm are executed in a parallel computing environment. Extensive experimental results show that the parallel implementation using simple adaptive partitioning methods can achieve substantial speed–ups, without significantly affecting the quality of the grouping. As discussed in Section 5, spatial partitioning is not only appropriate for the given application, but it is applicable to parallelize computationally expensive spatial analysis tasks. As it was demonstrated, SCSQ can easily accommodate the parallel implementation of such tasks.

Future work will be along four paths. First, for the adaptive partitioning methods, the effects of keeping a longer history versus sampling more frequently will be investigated. Second, the density–based and dual–objective spatial stream partitioning methods will be implemented and their effectiveness evaluated. Third, the proposed partitioning methods, independent of the rate of flow, always construct a fixed number of partitions. While not substantially, but as the number of partitions increases the grouping quality decreases. Hence, an adaptive partitioning approach in which the number of partitions is increased / decreased depending on the rate of flow will be devised and tested. Finally, to preserve clarity the paper presented the generic TG algorithm in its simplest form. In particular, in the presented version all vehicles are assumed to have the same passenger capacity and all requests have a common minimum savings parameter. Furthermore, in–route grouping, i.e., assigning requests to already active but not fully–occupied vehicle–shares, is not handled by the simple version of the TG algorithm. Future work will consider the implementation of a more complex version of the TG algorithm that addresses the above issues.

Acknowledgements

This work was supported in part by ASTRON and the Danish Ministry of Science, Technology, and Innovation under grant number 61480.

8. REFERENCES

[1] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, (18)9:509–517, 1975.

[2] J. L. Bentley and M. I. Shamos. Divide–and–Conquer in Multidimensional Space. In *Proc. of ACM–STOC*, pp. 220–230, 1976.

[3] CARLOS Ride–Sharing System. `http://www.carlos.ch/`

[4] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, et al. Scalable Distributed Stream Processing. In *Proc. of CIDR*, 2003.

[5] T. G. Crainic, F. Malucelli, and M. Nonato. Flexible many–to–few + few–to–many = an almost personalized transit system. In *Proc. of TRISTAN*, pp. 435–440, 2001.

[6] R. Finkel and J.L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. In *Acta Informatica 4 (1)*, pp. 1–9, 1974.

[7] M. Ivanove and T. Risch. Customizable Parallel Execution of Scientific Stream Queries. In *Proc. of VLDB*, pp. 157–168, 2005.

[8] G. A. Frank and D. F. Stanat. Parallel Architecture for k–d Trees. Technical report, North Carolina University at Chapel Hill Dept. of Computer Science, May 1988.

[9] M. Gebski and R. K. Wong. A New Approach for Cluster Detection for Large Datasets with High Dimensionality. In *Proc. of DaWaK*, pp. 498–508, 2005.

[10] G. Gidófalvi and T. B. Pedersen. Spatio–Temporal Rule Mining: Issues and Techniques. In *Proc. of DaWaK*, pp. 275–284, 2005.

[11] G. Gidófalvi and T. B. Pedersen. ST–ACTS: A Spatio–Temporal Activity Simulator. In *Proc. of ACM–GIS*, pp. 155–162, 2006.

[12] G. Gidófalvi and T. B. Pedersen. Cab–Sharing: An Effective, Door–to–Door, On–Demand Transportation Service. In *Proc. of ITS*, 2007.

[13] A. Guttman. R–trees: a dynamic index structure for spatial searching. In *Proc. of SIGMOD*, pp. 47–57, 1984.

[14] T. Hägerstrand. Space, time and human conditions. In Dynamic allocation of urban space, ed. A. Karlqvist et. al. Lexington: Saxon House Lexington Book, 1975.

[15] Hitchstrers. `http://www.hitchsters.com`

[16] C. S. Jensen, D. Pfoser, and Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. In *Proc. of VLDB*, pp. 395–406, 2000.

[17] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser. StreamGlobe: Processing and Sharing Data Streams in Grid–based P2P Infrastructures. In *Proc. of VLDB*, pp. 1259–1262, 2005.

[18] B. Liu, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In *Proc. of VLDB*, pp. 1338–1341, 2005.

[19] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[20] Transportation Problems. `http://www.di.unipi.it/optimize/transpo.html`

[21] Taxibus – Intelligent Group Transportation. `http://www.taxibus.org.uk/index.html`

[22] texxi – Transit Exchange XXIst Century. `http://www.texxi.com`

[23] E. Zeitler and T. Risch. Processing high–volume stream queries on a supercomputer. In *Proc. of ICDEW*, pp. 144, 2006.

[24] E. Zeitler and T. Risch. Using stream queries to measure communication performance of a parallel computing environment. In *Proc. of ICDCSW*, pp. 65–74, 2007.

[25] Y. Xing, S. B. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *Proc. of ICDE*, pp. 791–802, 2005.