Non-Intrusive Contact Sharing and Horizontal Database Partitioning

Mac Finnie

Distributed Systems - CS 339
Professor Jeannie Albrecht

Williams College

Abstract

We build a distributed system for sharing contact information based on agreement via simple gesture (synchronized button press) with goals of expanding to more complex gestures such as a handshake. We integrate a Pebble Smartwatch, Android OS Smartphone, and a back-end server as hardware components. A contact exchange is initiated by a synchronous button press on the watch. The watch notifies the phone that a button-press gesture has been detected, and the phone includes information about time and location into an HTTP Get Request to the back-end server. In this way the watch serves as a sensor, the phone a relay, and server a matchmaker. Additionally we address scalability by implementing and testing a horizontal partitioning of the back-end database by various heuristics. This paper's role is both as an exposition of the system's components and design, and as a directory of possible directions in which the project could expand.

1 INTRODUCTION

This project aims to build a system to automate the exchange of contact information between two individuals. Firstly, let us clarify the spirit in which this system is designed.

Designing a distributed system involves organizing disparate elements into a cohesive, *functioning* unit. From the point of view of a distributed systems designer, a piece of hardware is as much a member of a greater whole as it is an individual. In this light, the world we live in becomes very interesting. Distributed activity amongst humans, for example, becomes strikingly apparent. In particular, one sees humanity's exponential rise in knowledge and technology as the production of many cooperative systems working in parallel. One may frame the whole of human society as a complex distributed system - existing through time, constantly gaining and losing nodes, educating existing nodes - whose original function of survival and reproduction has expanded into the strange and beautiful construction we see today.

In addition to being a powerful tool, interconnection is nearly ubiquitous in modern society - it is increasingly difficult to be self-reliant. Even if an individual were to achieve complete disconnection from society, they would still rely heavily on distributed systems of tools to achieve tasks. Therefore the systems one designs are for a world which is inherently distributed. In particular, the world contains systems which provide a service to individuals - e.g. shopping centers, transportation networks - and those which combine resources into a product or idea (e.g. farms, design teams, mathematics faculty).

When looking at *services*, one finds many systems which use computers to approximate an existing distributed service. A basic example is the reduction of brick-and-mortar retail shopping from requiring a system involving many pieces (e.g. transportation, building, customer service, etc.) and a time-consuming algorithm (i.e. the process of using the system's components to achieve "shopping") to one involving only a few pieces of hardware, a bunch of wire, and computer code. The replacement of brick-and-mortar retail by online retail is simple in that the "shopping" function does not require a high degree of cooperation between humans to complete - of course, it actually takes an immense amount of cooperation if we consider not only the original elements in the brick-and-mortar shopping system, but also the elements in the system involved in making that original element, and so on - however, the same can be said about the online replacement, so we move on. In other words, interactions occurring between humans in the shopping system are simply encoded exchanges of information (e.g. "Can I help you find anything?" - "Yes, where could I find the exit?" - "Right this way, sir"). So, in the world there are many distributed functions like shopping occurring in some system like a mall, with nodes communicating via simple exchanges of encoded information, and waiting to be replaced by computer systems like Amazon.

Likewise, the generation of many *products* has become computerized. For example, consider the production of an automobile. This is a function which previously required many people working in unison to complete. However, Henry Ford gave us the assembly line and with it a simple encoding of each worker's task in the system. Therefore, once advances in robotics allowed for the automation of tasks once requiring a human's dexterity, the system of producing a car became automated distributed system.

Now turning to more abstract productions of humanity, like mathematics and artwork, it may seem that a computer system may be suited to take the place of faculty producing new theorems or a symphony performing Bach, but here we run into trouble. Not only is it unclear how an individual performs these functions, but it is also unclear how groups of individuals cooperate in the process. Our focus is here, in how we may learn more about how groups of people interact to produce some output.

Our project is an attempt to digitally construct the most basic and preliminary stage of human cooperation - the meeting of two people.

This project automates a simple human activity: *the exchange of contact information*. This activity rests in the domain of understanding human-to-human interaction. If we were to view humanity as a distributed system, automating the exchange of contact information is akin to describing the way in which nodes become aware of each other in a network. An understanding of this process is fundamental to the automation of functions as complex as those carried out by humans. Eventually, the project aims to become a starting point in augmenting existing collaborative systems by digitizing information on how individuals in a group interact. Such information is vital in improving the various systems which people take part in, an area which has seen less development than services for individuals.

In abstract: fundamental to a system in which agents must communicate to achieve a goal is the moment when two individuals become aware of one another. That is, a formal exchange of information about the individuals. In this way, each agent in a group becomes aware of some subset of the entire population, or becomes connected to some subset of the entire network. And, if we follow the work of sociologists and the "Six Degrees Principle" (in which it is theorized that individuals are separated by at most 6 hops in the social network graph) then our society becomes a densely interconnected social network. Understanding how this network becomes interconnected is a fascinating question.

Specifically this project is,

A system to automate the exchange of contact information between two individuals based on a synchronous gesture.
A partitioner to format the supporting database and improve performance and scalability

2 AUTOMATION

Before describing our system, a few words could be said about what it means to automate a function. We define an *automated system* as one which has incorporated computer systems to perform tasks previously done by humans. Likewise, a *fully automated* system is one in which no human involvement is necessary. Any process which involves humans can be automated.

For example, websites like Amazon have automated the shopping service. As with many services, the human element is not completely removed and thus the system is not fully automated. Essentially, a system like online retail is designed to allow for quicker execution of the shopping function by a person. Though trending in that direction, such systems have not yet fully *automated* the function. That is, we do not yet receive all of the goods we desire exactly when we desire them. It is important to note that it is not exactly clear if full automation of services like shopping is desirable - perhaps the experience of using the service is what we value.

In order to fully automate a process such as shopping, one would have to be collecting information from a person on what it is they are interested in purchasing at any moment in time. In particular, one would look to the ways in which that person encodes that information in the world (whether intentional or not). An example is the targeted advertising project we completed this semester.

In this project, we look for the ways in which a pair of individuals encodes an exchange of contact information. Many cultures have a gesture for such an exchange - a common example is the handshake. If one was able to detect a handshake, then one could initiate an exchange of contact information without ever requiring the user to act. In this way the system would be fully automated.

Though detecting a gesture like a handshake is a goal of future work, we use instead the detection of a button press to initiate an exchange. This requires user action, and is thus not fully automated, but reduces the amount which the user must be active. For example, using our system eliminates the need for a user to go through the stages of requesting contact data from the other individual and then entering that information into a phone application.

3 DEVELOPMENT SUMMARY

If I have time, there will be a few notes here about developing using Pebble and Android's development environments. It is interesting to think about when programming for a mobile platform becomes a mobile activity itself. That is, what is the relationship between the ability to write, compile, and install new software at the level of mobility of the platform you are programming for.

4 ARCHITECTURE SUMMARY

We refer to the attached architectural overview diagram.

The system has three main hardware components:

$$\begin{aligned} WATCH(w_i) &= PebbleSmartwatch \\ PHONE(p_i) &= AndroidOS \\ SERVER &= Sysnet0 \end{aligned}$$

Likewise, the system has three main software components:

$$\begin{aligned} sensor.c &=> [WATCH] \\ relay.js &=> [PHONE] \\ matchmaker.php &=> [SERVER] \end{aligned}$$

Launch: This is a description of how to use the system, and will hopefully give some intuition as to the operation and organization of the hardware and software elements. In development, installing the application onto the Pebble Smartwatch one needs to log into $cloudPebble.net/ide/$, an online code editor which Pebble provides. Once the app is installed, it can be updated at any time by re-installing via this same method. To run the application, navigate to the list of apps on the Pebble and select the application named "Contact Sensor" (note that upon installation the application is automatically launched).

You are greeted by a message which says "Waiting for JS to load" (we explain this in "Details" section). Once the Javascript located on the phone has loaded, the screen displays "Sensing for contacts". At this point, the system is ready to be used. To share a contact, find another user of the application and press the "select" button (the center button on the right side of the watch) in tandem. The system will then match your recorded button press with your partners, and contact information will be shared. If a button press is not in sync with another's button press, then no contact information is shared.

Operation: Now, we explicitly name the components involved in the system's execution. As a list, the system works as follows,

I: Button press detected ($sensor.c$)
II: Notification message sent to phone ($sensor.c \rightarrow relay.js$)
III: Form a record of the event ($relay.js$)
IV: Send XMLHttpRequest to Sysnet0 requesting match ($relay.js \rightarrow matchmaker.php$)
V: Insert data into database ($matchmaker.php$)
VI: Select nearest entry from the database ($matchmaker.php$)
VII: Return to phone ($matchmaker.php \rightarrow relay.js$)
VIII: Return to watch ($relay.js \rightarrow sensor.js$)
IX: Notify user ($sensor.c$)

Forgiving repetitiveness, the system as a pipeline is : $sensor.c \rightarrow relay.js \rightarrow matchmaker.php \rightarrow relay.js \rightarrow sensor.c$.

Looking at the table of operation and the pipeline, the system is revealed to be a basic example of a distributed system. That is, the messages follow a simple path involving three nodes and two edges. I do not know anything about network theory, but this feels like a classified network topology. We will mention the impact of system with multiple users on the network topology.

Version Notes: The project has had three main forms in its history. The first version was the initial attempt to detect a handshake - minimal work had been done on an Android application, javascript and a back-end server component had not been realized. The second version developed an Android application in java, and began to develop a back-end server. The third version, which this final project presents, adds a javascript program to relay information from watch to server.

The watch contains an accelerometer which can be used to collect data and infer a user's gestures. We do not use the accelerometer in the current version, but rather use the center button on the right side of the watch (known as the "select" button to Pebble). A press of the select button is listened for, and upon being detected a notification is sent to the javascript program $relay.js$.

In this version of the project, we are not using the Android application for contacting the Sysnet0. We use instead use a javascript program which also runs on the phone. The reason for this switch is for simplicity, as Pebble provides a library called PebbleKitJS providing functions for communicating with the watch. Using $relay.js$ is also helpful because javascript provides simple methods for accessing gps location and time - the current data we base pairing on.

The final details are about the operation of the server script, $matchmaker.php$. The server is responsible for receiving event records from users and returning either a found partner or a notification that no match was found. We chose PHP because it is designed to be closely integrated with $MySQL$ Databases. This cohesion is manifested in a library of PHP functions called $mysqli$. This standard library makes it simple to connect to databases (either remote or local) and to execute SQL queries.

$MySQL$ was originally chosen because of it being a server in itself. That is, it "serves" clients which wish to access its data by requiring a connection. One provides a username, password, and tablename to connect. We thought that such a server could be connected to remotely (from the phone) using $JDBC$ in the java application. This turned out not to be a reliable possibility with Android's version of JDBC. But, it turned out that needing the database to be web connected meant that we needed a web script to serve information from the database. The integration of PHP and $MySQL$ prompted us not to change.

5 DATABASE PARTITIONING

The main addition of our final project was a program for partitioning the back-end database. It is implemented as ***partitioner.php*** and utilizes built-in features of SQL and PHP's strong integration with the database to partition its main table into smaller more focused regions. The smaller regions make queries to the database which include some conditional statement (e.g. SELECT * FROM table WHERE id ¡ number) perform better because they are searching a smaller section of the data.

The motivation to partition the database comes from improving the scalability of this system. In our system, load can be measured as the number of users who are "using" the system at a point in time. That is, the ability to support many pairing requests is to be scalable. Therefore, to improve scalability we must improve the portion of the system where user traffic intersects.

All user traffic uses the $matchmaker.php$ script on $Sysnet0$. A system with many active users will contain many matchmaker scripts running concurrently on the back-end. Each of these scripts is accessing the database at two points: First to $INSERT$ the data received by a user, Second to $SELECT$ another active user whose record minimizes a distance function.

The performance of our system at these high levels of concurrency depends upon how $MySQL$ behaves when multiple queries are being submitted in sync. Firstly, $MySQL$ is a multithreaded database - a new connection will spawn a thread to handle transactions made by that connection. To dig deeper, we are interested in looking $MySQL$s ”storage engine” called $InnoDB$ - the engine which determines how access to tables stored in memory occurs. Most of my reading turns up that queries to $InnoDB$ are atomic, and thus guarantee consistency but lock out other threads during execution. $MySQL$ defines this ”locking” as, ”the system of protecting a transaction from seeing or changing data that is being queried or changed by other transactions.” in the $MySQL$ glossary.

This investigation of how our system behaves under increased load leads to the conclusion that as load increases there is an increased chance of being locked out of a database transaction. Being locked out requires retrying the query until it is successful. To reduce the effect of locking, and to add many other benefits to the system, we can ”partition” the database.

$MySQL$ supports partitioning of individual tables across a file system, and can be configured to use a specific directory for each database. We investigate the partitioning of a single table as opposed to symbolic linkage of multiple databases because we see partitioning as more logical first step in scaling the system. This comes from viewing the table as a child of a database in the $MySQL$ hierarchy. In other words, databases contain tables implies that databases are of a greater ”scale” than tables. Therefore, we focus first on partitioning a single table.

The table we partition is the $ACTIVE_EVENTS$ table which holds all unresolved user pairing requests. The partitioning is ”horizontal” in that it divides by row and not column. In a way, the decision of whether to partition by row or column feels like the difference between Google’s inverted index and forward index - i.e. from content to documents and from documents to content. But, I cant really construct a direct analogy at the moment. We then defined a partitioning function which assigns a row in the table to a partition of the filesystem by conditions on entries of the row. This was defined in SQL as,

PARTITON BY RANGE COLUMNS(HEURISTICS)(

PARTITION p0 VALUES LESS THAN(v0)
...
PARTITION p0 VALUES LESS THAN(vN)

);

When partitioning by $RANGECOLUMNS$, one must define a set of columns which serve as input to the partitioning function. We label this set $HEURISTICS$. The function itself is truly a list of conditional functions which collectively create a well-defined function from a row to a partition based on HEURISTICS values.

This is implemented as a PHP script which reformats the database by partitioning in a specified way. Reformat is not a technical term, but rather refers to reconfiguring the storage of a single table in memory. That is, creating an organization of the tables based on the heuristic used for partitioning. The script again utilizes PHPs close integration with $MySQL$. The $mysqli$ library allows us to run queries on the database, including $MySQL$s built in $PARTITION$. In particular the script does the following,

partitioner.php

I : Save backup of table.
II : Create new table with specified partitioning function.
III : Import data from old table to new.

Having partitioning as a PHP script is convenient because it opens up possibilities for a dynamically partitioned database so as to present the optimal partitioning. An ideal partitioning is most likely to be defined hierarchically, with the root as the entire table and splits by $HEURISTIC$. This could be implemented by keeping multiple tables, or by keeping a list of the partitions $\{p_0, ..., p_n\}$ which we organizer in a tree data structure within $matchmaker.php$ and accessed accordingly. Nodes on this tree could be formed by union of the partitions. As one can specify which partitions to look in when executing a $SELECT$ query, this would allow a mechanism for accessing hierarchically organized data.

6 EVALUATION

There are many possible parts of our system which would be interesting to evaluate. We would like to know the accuracy of the current distance function, we would like to know the speed of using Javascript and $PebbleKitJS$ vs. using the Java Android Application, but most fitting for this paper is a test of the effectiveness of partitioning the back-end database.

We used *sysbench*, a benchmarking tool used in analyzing database and file i/o performance. The reason for using this was that it allowed an easy mechanism for running many test queries in parallel. Sysbench is quite limited in features, where it gives many options but not much customizability. For example, the table it uses as test is fixed in its column names and datatypes. In this way sysbench provides a mechanism for testing, but not for truly *simulating* our system. Originally, we set out to build a script that would simulate various loads on various partionings of a database. We got quite far into that development, but ran into trouble and out of time and energy. We will describe the difficulties involved with simulation, and propose our original approach.

In actuality, our tests were evaluated using sysbench as follows,

test schematic

pre: Create 3 databases: sys1,sys2,sys3 to represent a non-partitioned, 2-partitioned, and 5-partitioned database table structure.
I. Build a table on each database with TRAFFIC number of entries using Sysbench perform function.
II. Run the partitioner to create partitionings of sys2 and sys3 tables.
III. Run sysbench and collect results.

The results of this testing are attached as a table and as a series of graphs. Next is a bit on our attempt to simulate.

In general we are testing to investigate the effect of changes in the variables: HEURISTIC, DEPTH, TRAFFIC - the fields on which we determine partitioning, the number of layers in a table partitioning, and the number of pairs using the system respectively. The tests would be run on unix server ($Sysnet0.cs.williams.edu$) using a PHP script ($evaluator.php$) to run the following testing algorithm,

evaluator.php

I : Partition the database by HEURISTIC and DEPTH
II : Generate as many pairs as TRAFFIC
III : Order by time field.
IV : Somehow reflect this time in the launching of user requests to use matchmaker.php
V : Compute average $time\_to\_insert$ and $time\_to\_select$ in matchmaker.php

We would run $evaluator.php$ on the following values of $HEURISTIC$, $DEPTH$, and $TRAFFIC$:

$$HEURISTIC = \{TIME, LOCATION\}$$
$$DEPTH = \{0, 1, 5, 10, TRAFFIC\}$$
$$TRAFFIC = \{1, 5, 10, 50\}$$

A test is a simulation of roughly 2 minutes of system activity.

Each simulation contains as many pairing requests as the value of the variable $TRAFFIC$.

A pair has the form, (a , atime , alat , alon) (b , btime, blat, blon)

Possible values of time,

($SEEDTIME$ - 60) ¡ time ¡ ($SEEDTIME$ + 60)

Possible values of location,

41000 $< latitude <$ 44000
75000 $< longitude <$ -72000

This defines a grid around Williamstown. We skew the data to always reflect actual matches. To simulate this scenario we,

I: Generate time, longitude, latitude offset FOR THE PAIR

II: Generate time, longitude, latitude offset FOR THE INDIVIDUAL

In this way we have a two event records which are slightly varied around a central location and time of the pair's encounter. Testing "noise" in the system will be important to further evaluation.

7 FUTURE WORK

The future is bright for this project. In the very limited functional progress I have made there have been opportunities to learn. It was an introduction to web development beginning with the smartwatch - quite an interesting trajectory.

Now, as the system begins to take shape, there are many different focus-areas more work can be done on. In particular, there are three main categories of development for the three main functional components of the system.

Firstly, there is developing with the smartwatch in mind. We used a Pebble Smartwatch, but utilized it as a simple sensor relay. Thus, what other hardware components could replace the watch? And, is the watch an "end" as considered in the "End-to-End Argument" - does that determine what amount of work should be done at this point of the system?

Secondly, we have the phone application. The code needs to be organized, but even then does not provide much outside of a simple UI and communication with the watch. There is work to be done here in the application itself. The application has the power to bring the entire project into vastly different places. How the user interacts with the components of the system in order to us it is largely orchestrated by the phone application.

Lastly, there is the component which I worked most heavily on in this final project. The back end server is a great example of a widespread technique for serving data. Work on the server can almost be split into two pieces: 1. work on the web element, receiving user requests 2. work on the database element, fulfilling user queries. I have already spoken about some of the directions in which further study of partitioning the database could head. An important aspect of research here is in realizing the importance of the natural structure of data, and how finding that structure can lead to more efficient modes of storage and access. There is new ground in reflecting upon the connections between how the data of a system can be best visualized, stored, collected, and most efficiently accessed.

The depth of this research comes from its combination of interesting systems problems and social implication. Most generally, we are exploring how one might collect information from an agent's activity in the world in order to automate a function. In particular, we are interersted in collecting information from activity resulting from an *encounter between agents* which results in the execution of a common function. For us, of course, the agent's are human beings, the activity is a gesture, and the function is storing the other's contact information.

I will write more in a README of the organized source for the entire project once I get time to organize it.

8 CONCLUSIONS

Our tests verified the speed up at scale associated with partitioning a database. However, we have only scratched the surface of tuning how we access and store data in the database. I learned that database organization is a very rich area of discussion, as databases sit behind most internet enabled companies.

Partitioning the database is a necessary optimization during times of high load. In particular, occurrences of database locking increases with the number of transactions. Furthermore, noticing that our system is a method of pairing - that is each query performs a function to find *a single* other entry - we see that there is even greater incentive for small personalized table partitions. That is, in theory a matchmaker request needs access to a table with only a single entry - that of the user's closest partner. So, as near to that ideal as we can get, the more efficient each database transaction will be. In this way, our project is designed to be scalable.

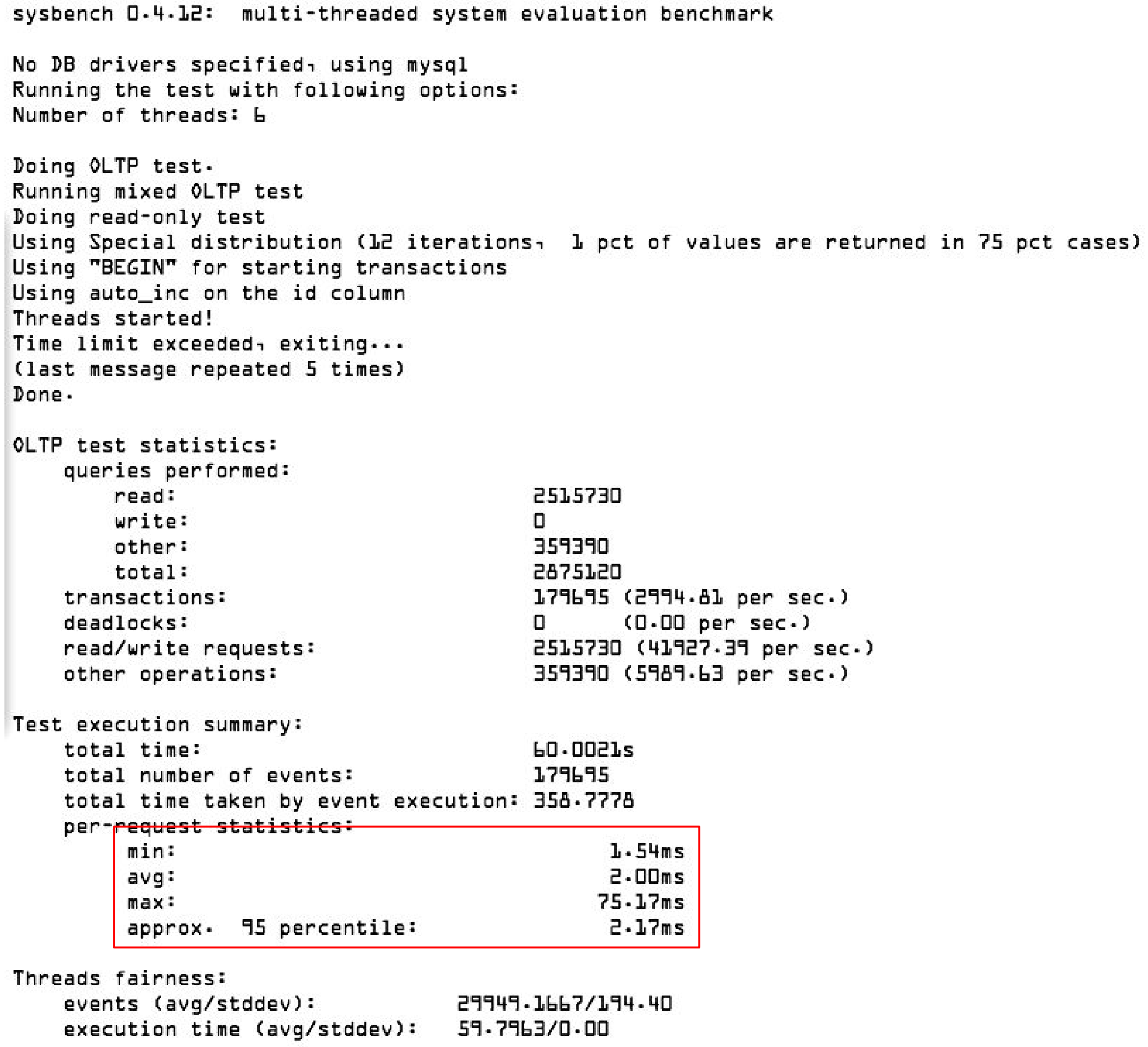
Partitioning a database is fairly well documented online, specifically on MySQL and InnoDB's documentation pages. It is clear that use of such a technique is widespread, and it may be interesting to read more about systems which do so.

An interesting lesson is that when creating a strict format for a system's data storage, one often dictates system features. For example, partitioning our main table made it necessary to provide the parameters to do so. One possibility was to create a new table for each day and then to partition that table by TIME so that at any given point in time queries are only being made on the most relevant data. That heuristic does not do much to create an optimal partitioning. This is most obvious in the fact that all users are still querying from the same partition of the database at a given point in time.

Therefore, we needed to find a more personalized HEURISTIC with which to partition the database. Partitioning is, in our case, the grouping of user's data together into parts which are most likely to contain pairs. There must be a discernment *between* users and an establishment of a partitioning function *before* any pairing occurs, so that a new entry can be assigned to a partition. But how do we do this initially - before we have any users? We implement a static grid of location tuned to Williamstown, but recognize that this is an area of the system which ought to become dynamic. As we know, the database should be partitioned to pool the most likely pairings together in the smallest table partition possible. In order for this to be done we must add a mechanism for responsiveness of the database structure to the current user population.

Non-Partitioned
1000 row table

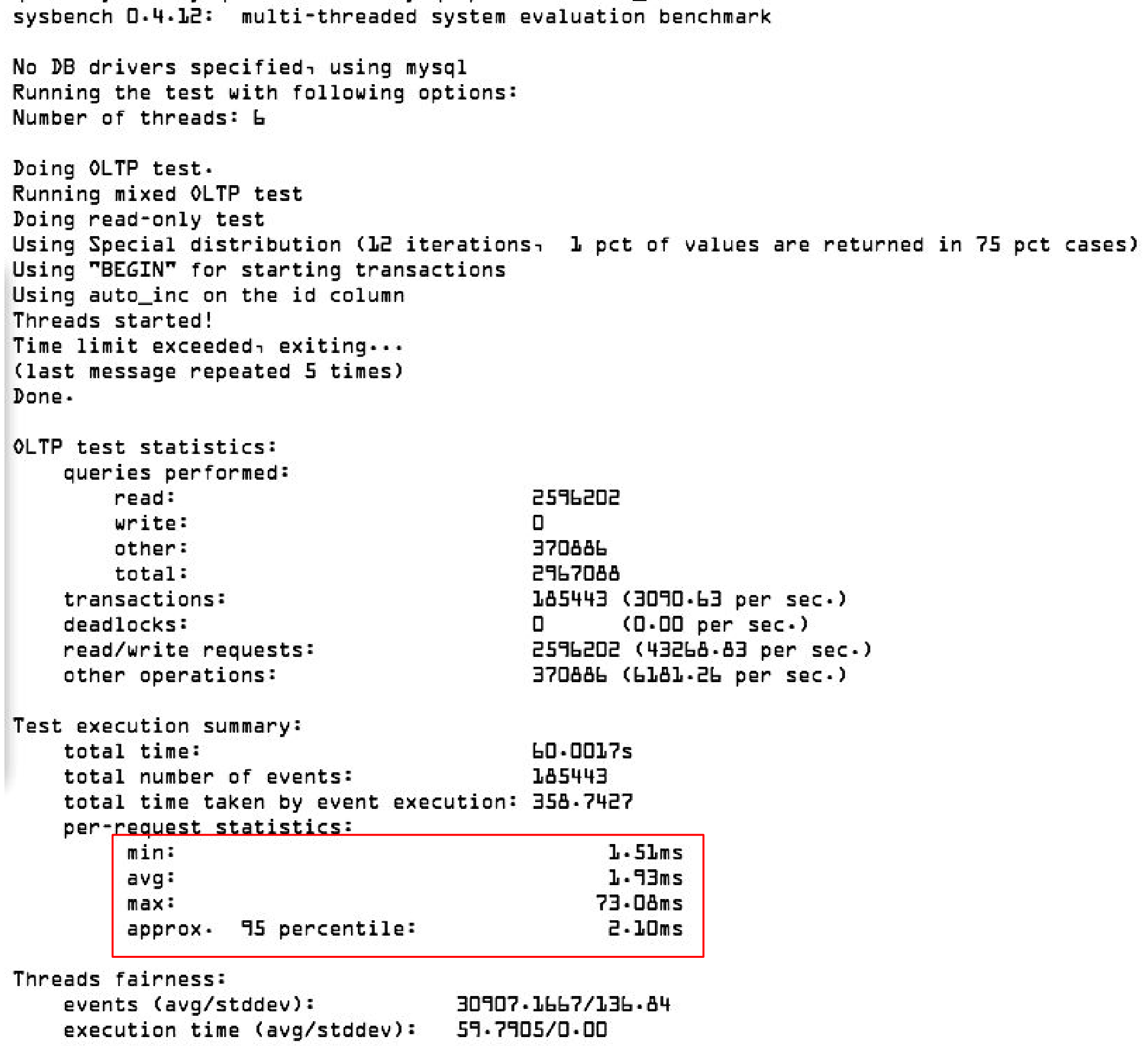
Read-only



```
sysbench 0.4.12:  multi-threaded system evaluation benchmark

No DB drivers specified, using mysql
Running the test with following options:
Number of threads: 6

Doing OLTP test.
Running mixed OLTP test
Doing read-only test
Using Special distribution (12 iterations,  1 pct of values are returned in 75 pct cases)
Using "BEGIN" for starting transactions
Using auto_inc on the id column
Threads started!
Time limit exceeded, exiting...
(last message repeated 5 times)
Done.

OLTP test statistics:
    queries performed:
        read:                            2515730
        write:                           0
        other:                           359390
        total:                           2875120
    transactions:                        179695 (2994.81 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 2515730 (41927.39 per sec.)
    other operations:                    359390 (5989.63 per sec.)

Test execution summary:
    total time:                          60.0021s
    total number of events:              179695
    total time taken by event execution: 358.7778
    per-request statistics:
         min:                                  1.54ms
         avg:                                  2.00ms
         max:                                 75.17ms
         approx.  95 percentile:               2.17ms

Threads fairness:
    events (avg/stddev):           29949.1667/194.40
    execution time (avg/stddev):   59.7963/0.00
```

2-Partitioned
1000 row table

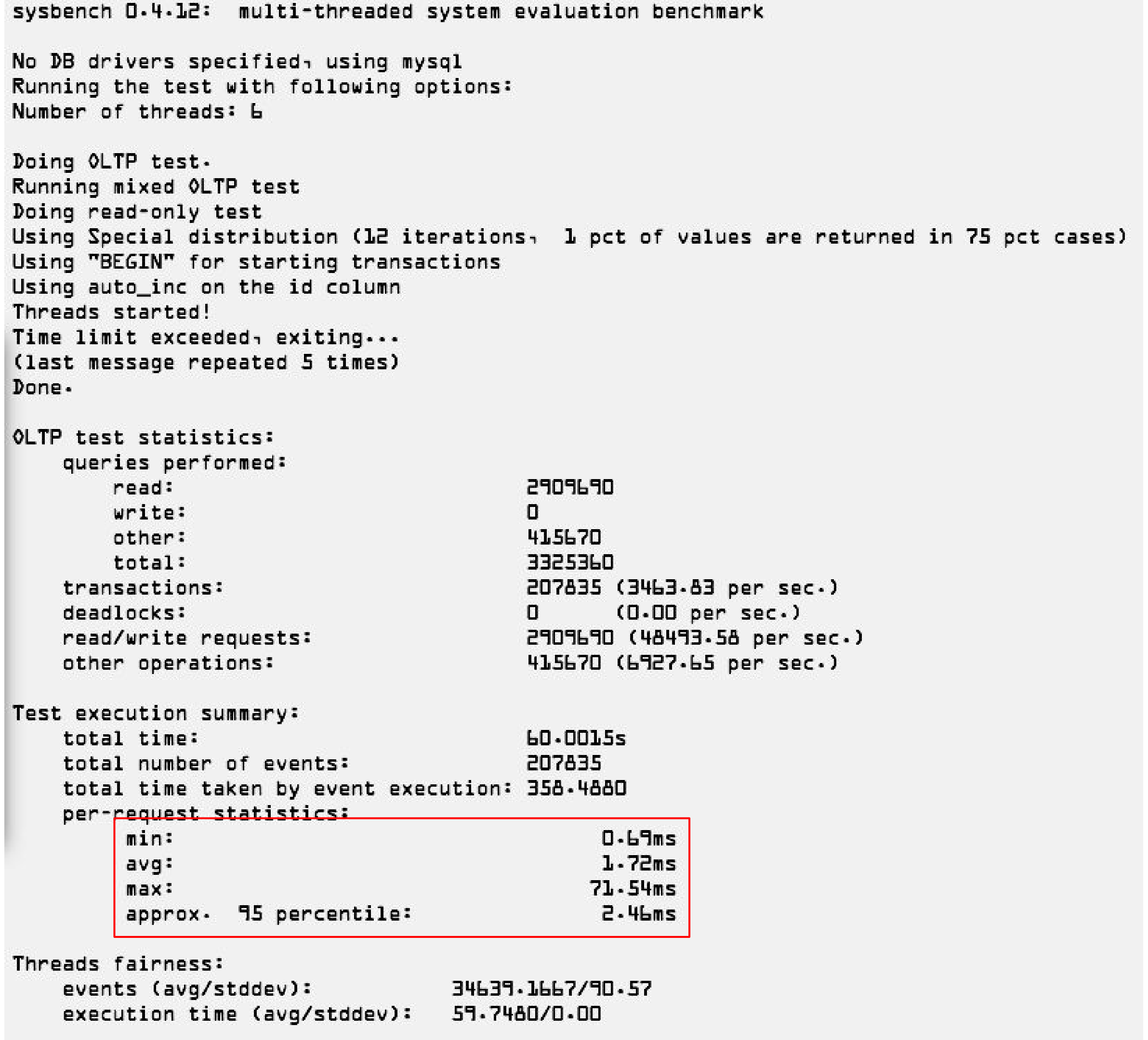
Read-only



```
sysbench 0.4.12:  multi-threaded system evaluation benchmark

No DB drivers specified, using mysql
Running the test with following options:
Number of threads: 6

Doing OLTP test.
Running mixed OLTP test
Doing read-only test
Using Special distribution (12 iterations,  1 pct of values are returned in 75 pct cases)
Using "BEGIN" for starting transactions
Using auto_inc on the id column
Threads started!
Time limit exceeded, exiting...
(last message repeated 5 times)
Done.

OLTP test statistics:
    queries performed:
        read:                            2596202
        write:                           0
        other:                           370886
        total:                           2967088
    transactions:                        185443 (3090.63 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 2596202 (43268.83 per sec.)
    other operations:                    370886 (6181.26 per sec.)

Test execution summary:
    total time:                          60.0017s
    total number of events:              185443
    total time taken by event execution: 358.7427
    per-request statistics:
         min:                                  1.51ms
         avg:                                  1.93ms
         max:                                 73.08ms
         approx.  95 percentile:               2.10ms

Threads fairness:
    events (avg/stddev):           30907.1667/136.84
    execution time (avg/stddev):   59.7905/0.00
```

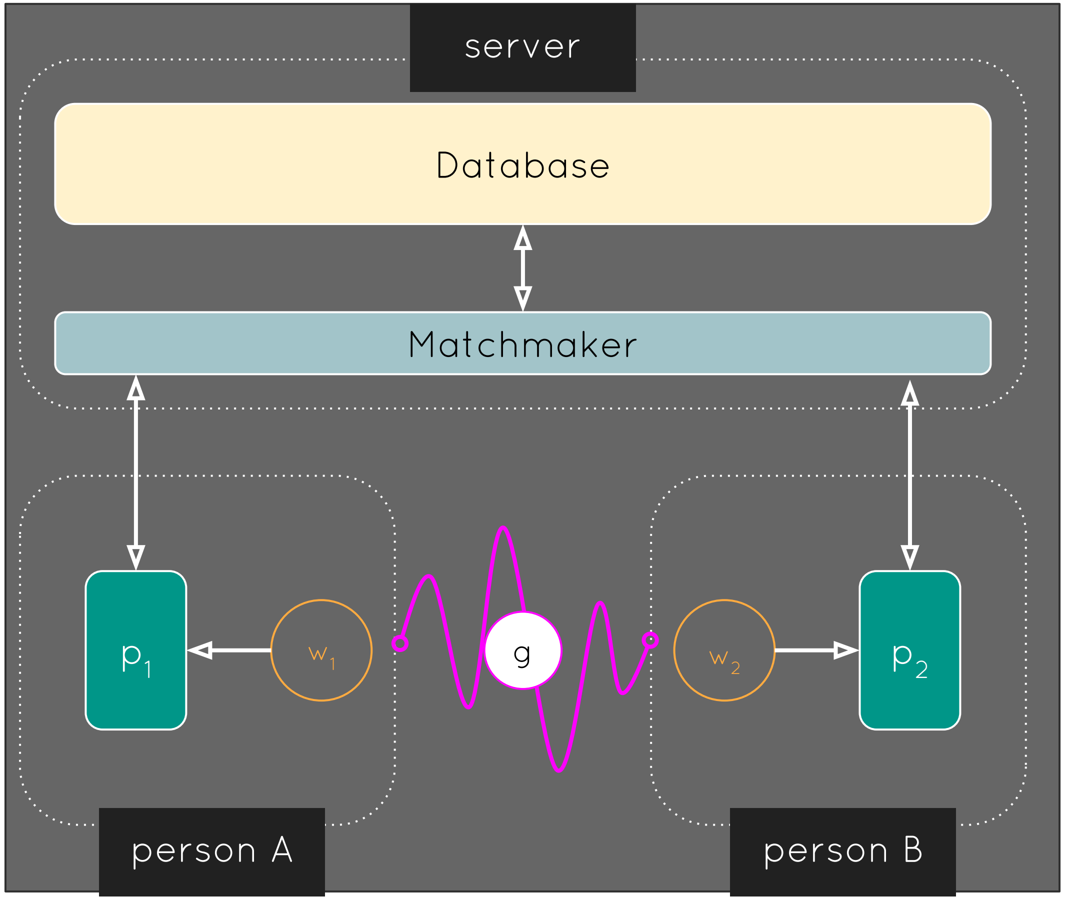
5-Partitioned
1000 row table

Read-only



```
sysbench 0.4.12:  multi-threaded system evaluation benchmark

No DB drivers specified, using mysql
Running the test with following options:
Number of threads: 6

Doing OLTP test.
Running mixed OLTP test
Doing read-only test
Using Special distribution (12 iterations,  1 pct of values are returned in 75 pct cases)
Using "BEGIN" for starting transactions
Using auto_inc on the id column
Threads started!
Time limit exceeded, exiting...
(last message repeated 5 times)
Done.

OLTP test statistics:
    queries performed:
        read:                            2909690
        write:                           0
        other:                           415670
        total:                           3325360
    transactions:                        207835 (3463.83 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 2909690 (48493.58 per sec.)
    other operations:                    415670 (6927.65 per sec.)

Test execution summary:
    total time:                          60.0015s
    total number of events:              207835
    total time taken by event execution: 358.4880
    per-request statistics:
         min:                                  0.69ms
         avg:                                  1.72ms
         max:                                 71.54ms
         approx.  95 percentile:               2.46ms

Threads fairness:
    events (avg/stddev):           34639.1667/90.57
    execution time (avg/stddev):   59.7480/0.00
```

architectural overview


server
Database
Matchmaker
p1
w1
g
w2
p2
person A
person B