Benchmarking Database Management Systems for Communication Cost Analysis

W Anthony Young - 20161423

wayoung@uwaterloo.ca

May 10, 2005

Abstract

Federated database systems are a useful tool for businesses and researchers around the world. These systems allow data from multiple remote data sources to be logically combined into one unified local data source. Using this system, queries that would traditionally require query fragments to be submitted to multiple sites can be performed by submitting one query to a central site. This central site can make use of data stored at the different remote sources as though the central site were simply an application requesting data.

These so-called global queries must be optimized, but many additional factors combine to make global query optimization complicated. Beyond the problems of local query optimization, additional costs, including the cost of communication, the cost of remote site optimization, etc., must be factored into cost models. Currently, the performance of

global queries in iAnywhere Adaptive Server Anywhere is much worse than the performance of local queries.

This paper presents some background material on federated database systems including information regarding benchmarking performed at iAnywhere Solutions, Inc. during a cooperative work term. A discussion of the results of this benchmarking as well as some initial recommendations for improving the performance of global queries will be presented. Some future work is also outlined.

Acknowledgements

This research was supported by iAnywhere Solutions, Inc., 445 Wes Graham Way, Waterloo, ON, N2L 6R2. Special thanks are extended to Glenn Paulley and Ivan Bowman who answered countless questions and provided support and guidance. Thanks to Mike Demko, Anil Goel, Ani Nica, Dan Farrar and Matthew Young-Lai who provided additional support during the eight month cooperative work term. Thanks are also due to Mark Culp, Scott Shepherd and David Peynado for the use of their hardware during the experimental phases of this project. Thanks to Karim Khamis, Graham Hurst and Ian McHardy who provided their help and expertise when applications were not functioning as expected. Thanks to the rest of the iAnywhere team for their efforts to make the cooperative work experience a memorable one. Thanks are also due to Nathan Boersma for his editing and suggestions. Last but not least, thanks to Frank Tompa at the University of Waterloo for his help in securing a cooperative education placement and guidance in preparing this work term report.

Contents

List of Figures

List of Tables

1 Introduction

John is sitting at his desk at Credit Co. finishing a report regarding the Acme account. He needs to provide Acme with a consolidated report regarding its various financial holdings. Since John manages Acme's financial investments, he has to locate all of the company's holdings around the world. John has a painstaking task ahead of him. He must contact each firm that Credit Co. invests in and determine how much of Acme's funds exist there. John, however, has a powerful weapon: a *federated database system*. Through the federated system, John can query all the firms that Credit Co. invests in around the world at once.

This paper presents some background information about federated (multi) database systems, including an analysis of some global query optimization strategies. Recommendations for improving the optimization strategy used in iAnywhere Adaptive Server Anywhere (ASA) will be given based on the results of some benchmarking experiments that were performed.

Throughout this document, federated database systems (FDBS's) and multidatabase systems (MDBS's) will be used to refer to the same concept: a system for integrating multiple heterogeneous (remote) data sources into a global data source [26]. Such products combine data from multiple database management systems (DBMS's) and make it logically accessible as if it resided on one central system. This allows John to simultaneously see Acme's holdings at each firm around the world. John may believe he is looking at a single database, but is in fact looking at many consolidated databases; giving a global view of multiple remote data fragments.

1.1 Motivation

A FDBS is a powerful tool. As noted in [8, 11, 12], several factors motivated the development of federated database systems.

- Proliferation of Heterogeneous DBMS's Within an Organization: It is not uncommon for different departments within an organization to have their own database servers. Departments may not coordinate to ensure that a corporation is using a homogeneous DBMS to store data. There is also no guarantee that the schemas individual departments use to store data will be homogeneous.

- Data Sharing Within Organizations: Many organizations seek to share data between departments. For example, the finance department may require information regarding projects in progress in the marketing department. Such information sharing is difficult without the guarantee of schema, model or language homogeneity.

- Differing Rates of Technology Adoption: Different departments will adopt technology at different paces. The information technology department is usually tech savvy. They will typically adopt new technology rapidly, while the human resources department often lags behind. Human resources may adopt technology slower as they use numerous paper forms that are hard to replace efficiently with electronic counterparts. Further, systems that are deployed at different times might come from the vender that is considered the "market leader" at the time of purchase.

- Mergers and Acquisitions: When companies join forces, their information systems must be linked. Applications used by individual business units will depend on old (pre-

merger/pre-acquisition) systems. Users may be reluctant to learn an entirely new system. With a FDBS, the systems can be merged logically. Users are still able to use their old applications and access languages. The merged systems can be viewed as one when this is required by a new application.

- Geographic Separation of Teams: Development teams may be broken up across geographic locations. A company may have engineering teams located in Ontario and California. Each site will host teams working on different projects, and may have its own information technology staff that makes its own purchasing and installation decisions. Often, there is no coordination between the information technology departments across sites. There are, however, situations when databases need to be queried across sites. Human resources may need to know what projects each employee works on regardless of their site location.

A FDBS is clearly a very useful tool for global businesses and researchers.

1.2 Organization

It is important for the reader to understand how data is organized and accessed in a federated system. This will help to illustrate how queries can benefit from optimization.

As seen in [19, 30], there are two main approaches to organizing data in a federated database system. The *multidatabase language approach* makes use of a specialized multidatabase language to submit and optimize queries. Thus the burden of integration is left to the system users and the remote database administrators. The *global schema approach* makes use of a global database schema composed of a consolidated view of translated remote database

schemas. Thus integration falls to a global database administrator who defines the global schema and data transformations to be applied. Other approaches, including *pass-through querying* [25], etc., will not be discussed here.

1.2.1 Multidatabase Language Approach

The multidatabase language approach uses a specialized query language to submit and optimize global queries [6]. The burden of integration is left to the system users and, to a lesser extent, the remote database administrators. Remote database administrators must provide schemas and semantic information about the data stored at their sites. Users must then formulate their queries to explicitly access data from remote sources. With a multidatabase language, users have much more control over the data they are querying. The heterogeneity of remote DBMS's can be exploited to allow *functional compensation* for language features implemented by one system that are not implemented by another. For example, some systems may not implement ranking; but, data can be gathered and ranked at a site that implements ranking before being returned to the user.

Since no global schema or translations need to be created, there is no need for a global database administrator. Once the system is set up it can run without significant maintenance. The multidatabase language approach requires users to know a significant amount of information about remote DBMS's and the data they hold. Users must know the structure and semantics of each remote schema in the federated system in order to make use of it. Explicit use of remote sites is required when writing a query. This is because there is no global schema created that the user may form queries against. Queries must be formed against the many

remote site schemas directly. Users are also required to learn the multidatabase language or language extensions.

1.2.2 Global Schema Approach

The global schema approach makes use of a global database schema composed of a combined and translated view of remote database schemas [3]. Thus integration falls to a global database administrator who defines the global schema and data transformations/translations to be applied when submitting a query to remote sites (note that remote site administrators may also need to define some translations on data leaving their sites). With this approach, users see a unified view of all federated data, and do not need to know how data is represented at the sites they are accessing. The database administrator combines the remote schemas into a single global one and generates the translation rules between global and remote column names, relation names and data formats. Functional compensation can also be provided by the FDBS.

Although a global database administrator is required for this approach, the burden of integration is lifted from the user (i.e. users do not need to know semantic information about data at a remote data source). Unlike multidatabase language systems, a global schema needs maintenance over time as sites, databases, relations and columns are added or updated.

1.3 Introducing OmniConnect

ASA contains a module named *OmniConnect* (Omni). Omni allows *remote data access* to data sources external to the server with the use of *proxy tables* [25]. Using Omni, ASA can

act as a FDBS capable of providing functional compensation with data that is accessed from a remote data source.

Once the ASA database administrator has set up proxy tables, users may access remote data as if it were stored in ASA: following the global schema model. Omni also supports pass-through queries that allow a user to send a query directly to a remote site without any parsing or execution by ASA. The use of some SQL extensions is required for this feature: following the multidatabase language model. Omni can be viewed as a hybrid global schema-multidatabase language approach to federation.

Presently, performance of Omni queries is much slower than performance of queries that access data stored exclusively in a local ASA database [25]. The aim of this project is to increase the performance of Omni queries through changes to the current global optimization strategy.

The remainder of this paper is organized as follows: section 2 introduces some basic concepts and methods of performing global query optimization. Section 3 discusses various factors that can affect global query optimization and presents a working cost model. An outline of the four applications developed to complete this project is presented in section 4. Section 5 discusses the experimental setup and procedures used to gather data useful for making recommendations to improve the current global optimization strategy. Section 6 presents the results of the outlined experiments. Section 7 provides some summary remarks and section 8 discusses some further avenues for research.

2 Global Query Optimization

2.1 The Need for Optimization

As with a *local query*[1], optimization of a *global query*[2] is important to ensure that response time for it is minimized. This allows a user to be as productive as possible. Due to the decrease in performance experienced by global queries (over local queries), global optimization can be of even greater importance. There are several factors that combine to make global optimization complicated [5, 15, 16, 30, 31].

- Remote Site Autonomy: There are several types of autonomy:

    1. *Data Autonomy* - Remote database administrators have direct and absolute control over the schemas, data types, relationships, etc. at remote sites. This information cannot be modified in any way to make it more convenient for consolidation.
    2. *Design Autonomy* - Remote database administrators decide when and how to replicate and fragment data.
    3. *Communication Autonomy* - Each site decides whether or not to communicate with other sites in the federation and the FDBS.
    4. *Execution Autonomy* - Each site can determine how, when and whether to execute global queries, including query prioritization.

[1] A query operating over data that resides on (is local to) the system to which the query was submitted

[2] A query operating over data that may be local to the system to which the query was submitted, but that may also reside at remote data sources

- Remote Parameters: Remote cost parameters for remote data sources are not always available to the FDBS. For example, the FDBS may not know what indices are available for relations at remote sources (nor can they predict which access methods will be used to execute a given query, etc.) as catalog and statistic data is not always accessible by the federation; and some data sources do not generate or export a catalog at all.

- Translation: Queries must be translated on-the-fly to and from the remote data source's schema, query language, and data model. This requires additional query processing time; as translations built and maintained by a database administrator must be used.

- Heterogeneous Capabilities: Not all remote data sources have the same capabilities. As a consequence intermediate results might have to be shifted to sites that can provide functional compensation, lengthening processing time.

- Additional Costs: Cost-based optimization of global queries needs to take into consideration additional factors including transmission speeds, network loads, and remote source configurations. As with remote parameters, this information is not always available to the FDBS.

Overall, one cannot assume any control over or even the existence of cost parameters for a remote data source. As far as the remote data sources are concerned, the FDBS is simply another application requesting data.

2.2 Optimization Strategies

Many query optimization techniques for FDBS's have been proposed in the literature [2, 3, 4, 7, 9, 10, 14, 21, 27, 29, 30, 31, 32]. This section provides an overview of some of these algorithms.

2.2.1 Semijoin Optimization Algorithm

The semijoin algorithm was proposed by Brill et al [3] and assumes that the cost of data transfer through a network outweighs remote site CPU overhead. This algorithm seeks to reduce the size of relations required for a query at local sites before transferring data back to the FDBS for query execution. It consists of four steps:

1. Site Selection - A set of sites that will be used to perform a query is first chosen. This requires finding a set of minimal size that includes one copy of each local, remote, partitioned and replicated relation (i.e. each site holding a data fragment must be in the set, but only one replica of a relation must be in the set). Some sites may hold more than one relation required by the query, further reducing the size of the site set.

2. Remote Reduction - Simultaneously at each remote site in the chosen site set, each relation is reduced by performing selections and projections. Parameters used to perform these operations are taken from SELECT, WHERE and JOIN conditions in the original query. It might be possible to optimize the order in which site reduction queries are performed by exploiting network traffic and speed, CPU load at remote sites, etc. (i.e. submit queries over slow links or to slow sites first in an attempt to minimize their impact on overall execution time).

3. Global Reduction - The FDBS finds and executes an efficient sequence of semijoins that will reduce the set of records to be transmitted. The original proposal utilizes a hill-climbing algorithm to determine this set. Once the semijoins are performed, the smallest amount of data required to answer the query is ready for transport.

4. Assembly - The data is next transferred to one central site and the result set is generated. The result set is then returned to the user. It may be less costly to generate the result set at one central site and then transfer the data back to the user. It may also be less costly to assemble the result set at the user's site.

This algorithm exploits the capabilities of the DBMS's in the federation through functional compensation in the remote and global reduction, and assembly phases. Reduction also attempts to minimize the transmission overhead required to send data between sites. The reader must keep in mind that this algorithm relies on an assumption that communication cost is the dominating cost parameter, which may not be valid.

2.2.2 Replicate Optimization Algorithm

The replicate algorithm was also proposed by Brill et al [3] but assumes that CPU overhead at remote sites outweighs communication costs between them. Therefore, this algorithm seeks to transfer data between remote sites in order to exploit the differences in processing speeds of each system. It consists of four steps:

1. Site Selection - As with the semijoin algorithm, a minimal site set is chosen. However, instead of choosing only one replica for each relation, we include all replicas of the data. This allows us to run queries in parallel at each replica.

2. Data Transfer - Each relation is copied to each site where it is to be used to process a subquery, but does not already exist (i.e. if site 1 holds relation A and requires relation B, B is transferred to site 1). This may require composing fragmented relations into one large relation. After this step, each site should have a copy of the relations that are to be used to form the partial query result for which that site is responsible (as per the subquery sent to it by the FDBS).

3. Query Execution - Once each site has the data it needs to run its partial query, the queries are executed. Once this step is complete, each site should have a partial answer to the user's query.

4. Assembly - Finally, the partial answers are transferred from the remote sites and the final result set is created at the user's home site. The results are then returned to the user.

This algorithm exploits the varying hardware configurations of the DBMS's in the federation. An attempt is made to reduce response time to a query by performing it in parallel at different sites. The reader must keep in mind that this algorithm relies on the assumption that execution overhead outweighs communication costs, which may not be valid.

The two main differences between the semijoin and replicate algorithms are:

1. *Assumptions*: depending on which assumptions are valid at execution time, either algorithm could be used.

2. *Execution Location*: replicate queries are executed at remote sites in parallel while semijoin queries are executed at the FDBS (or another central site) once all the required

data has been reduced.

2.2.3 The Garlic Optimizer

The Garlic optimizer uses *wrappers* to gather cost information used for developing a query plan [7, 9, 10, 14, 21]. Wrappers communicate *query fragments* (pieces of the original query that are dispatched to remote data sources) to the remote sites and provide costs for those fragments to the optimizer. The goal is to allow Garlic to find a good plan without knowledge of the capabilities of the remote sources (i.e. the wrappers must ascertain the capabilities of the sites and provide a good cost estimate for any given query fragment). The Garlic optimizer uses a set of *strategy alternative rules* (STAR's) to generate and rewrite query plans. Plans consist of a set of *plan operators* (POP's) that comprise the query plan tree (i.e. sort, filter, scan, etc). A generic "pushdown" POP that encapsulates work to be done at a remote site (i.e. table scans, etc) is also included.

STAR's are "fired" over the query string to generate POP's and can be seen as grammatical production rules. STAR's generate cost and cardinality information using input from wrappers. The Garlic optimizer works in three phases:

1. Fire Access STAR's - "Access" STAR's are applied to enumerate plans that read data from a source. The plan space is then pruned in order to remove plans that have the same or weaker cost properties (i.e. remove plans of higher cost that will not provide some interesting order to future joins, etc.).

2. Fire Join STAR's - "Join" STAR's are applied to enumerate all plans involving joins. The plan space is then filled with all possible join orders: Garlic considers bushy plans

and left-deep plans. Bushy plans are considered because collocated data may make a bushy plan more efficient (i.e. a join could be performed cheaply at a remote source instead of sending the data back to the FDBS for joining). The plan space is then pruned in order to remove plans with equal or weaker cost properties.

3. Fire FinishRoot STAR - The "FinishRoot" STAR is applied to provide orderings, selects, projects, etc. that were not already completed by a remote site (i.e. the site did not have the proper capabilities or sufficient data to perform the operation). The plan with the lowest cost is chosen for execution.

Literature [10] has shown that this algorithm is able to produce plans of near-optimal cost. The Garlic optimizer can be seen as a hybrid semijoin-replicate algorithm as it attempts to reduce communication cost and exploit varying hardware capabilities throughout its three phases. As with the semijoin algorithm, an attempt has been made to reduce communication costs using the Access STAR's and pushdown POP. The analysis of bushy plans also allows the exploitation of varying hardware configurations as is attempted with the replicate algorithm. This optimization strategy has a large amount of overhead, making it difficult to justify for simple queries.

Numerous other global optimization algorithms have been proposed [2, 4, 27, 29, 30, 31, 32] but will not be discussed here. The three algorithms presented were developed specifically for federated query optimization, while the additional algorithms referenced are extensions to existing optimization strategies. As such, the three presented algorithms were thought to be the most interesting.

3 Inputs to a Global Cost Model

There are many factors that can affect optimization of global queries in a federation. This section provides an overview of some of the most pertinent of those factors. A critical evaluation of communication cost will be made, and a simplified cost model will be proposed to test assumptions made by the various algorithms.

3.1 Factors Affecting Federated Query Execution

$$Cost = C_{OPT} + C_{COMM} + C_{EXEC} + C_{SM} + C_{RF} \tag{1}$$

Equation (1) presents a high-level global cost model. The various parameters of this cost model are discussed below:

1. Cost of optimization =

$$C_{OPT} = O_E + O_R + O_H + O_C \tag{2}$$

Equation (2) presents some of the factors affecting the cost of optimization of a query. Optimization cost will appear at the FDBS as well as each remote DBMS. Optimization cost is affected by several factors including query rewrite, O_R, plan enumeration and pruning, O_E, application of heuristics, O_H, and cost-based optimization, O_C. Since it is simply another application to the remote data sources, the FDBS cannot exert any control over remote site optimization times. The FDBS can, however, control how much time is spent globally optimizing queries (i.e. deciding on join strategies, performing site selections, etc.) using an *optimizer govenor* [25] or *stopping condition* [24]. A governor allows an optimizer to uniformly sample plans in the plan space by limiting the amount

of time spent evaluating each individual subspace. Limiting the evaluation time ensures sampling of a query space does not limit the exploration of plans to one specific subspace. The intention is to provide a more optimal solution by visiting each subspace. Stopping conditions can be imposed to halt optimizaiton once a set amount of time has been spent on some operation or portion of optimization.

2. Cost of communication =

$$C_{COMM} = C_{LS} * a + C_S * b + C_{DS} * c + C_T * d + C_{PF} * e + C_{PS} * f + C_R * g \quad (3)$$

Equation (3) presents some of the factors affecting the cost of communication of a query. Appearing many times during the execution of a global query, this cost must be accounted for during all data transfers between sites or to the FDBS. Communication cost is affected by several factors including link speed, C_{LS}, data size, C_S, data source, C_{DS}, data type, C_T, prefetch status ('on' or 'off'), C_{PF}, packet size, C_{PS}, and processor speed, C_R. The variables $a - f$ allow us to model that each factor has some affect on the overall communication cost.

3. Cost to reformat data =

$$C_{RF} = R_T + R_W \quad (4)$$

Equation (4) presents some of the factors affecting the cost of reformatting data used by a query. This cost will appear once for each remote data source that is accessed during a global query. Reformatting cost is affected by several factors including the time needed to perform data translations, R_T, and the time, R_W, to transform a query from the

global data language/model/schema into the data language/model/schema of a remote data source.

4. Cost of subqueries and method calls =

$$C_{SM} = \sum_{s \in S} Cost(s) + \sum_{m \in M} M_E(m) \tag{5}$$

Equation (5) presents some of the factors affecting the cost of performing subqueries and method calls that are part of the global query. Subquery statements are available to the global optimizer. Estimation of the time it will take to execute each subquery, s, in the set of subqueries, S, is simply a recursive call to the global cost function. Since the query statement for method (or stored procedure) calls is often not available to the global optimizer, it can be difficult to estimate the time it will take, $M_E(m)$, to execute each method call, m, in the set of method calls, M. As demonstrated by Adali et al [1], M_E can be empirically determined using feedback from query execution. Lookup tables could be used to allow an optimizer quick access to the previous execution times for a given method, m.

5. Cost of query execution =

$$C_{EXEC} = E_R + E_J + E_F + E_S + E_D + E_A + E_I \tag{6}$$

Equation (6) presents some of the factors that affect the cost of executing a query. This cost will appear for each query fragment, including once for any functional compensation provided by the FDBS. Methods of estimating the cost of data accesses, E_R, sorts, E_S, joins, E_J, etc. have been discussed in the literature [20, 22, 23]. Unfortunately, it is often

System	Query	Location	Response Time (seconds)
System 1	SELECT * FROM CUSTOMER	Shared Memory	9.20
	SELECT * FROM CUSTOMER	Ethernet Network	19.41
System 2	SELECT * FROM CUSTOMER	Shared Memory	4.69
	SELECT * FROM CUSTOMER	Ethernet Network	21.51
System 3	SELECT * FROM CUSTOMER	Shared Memory	8.67
	SELECT * FROM CUSTOMER	Ethernet Network	17.35
System 4	SELECT * FROM CUSTOMER	Shared Memory	14.93
	SELECT * FROM CUSTOMER	Ethernet Network	30.82

Table 1: Results from Experiment 0

difficult to determine the access, sort and join methods a DBMS will use; making accurate calculation nearly impossible. Calculated or hard-coded costs for performing operations such as filtering tuples, E_F, eliminating duplicates, E_D, performing aggregation, E_A, creating indexes, E_I, etc. will be system specific and unavailable to the global optimizer. As a result, it may be useful to obtain an estimate of query fragment execution cost from the remote data sources directly. This, however, might not always be possible as ODBC data sources such as text files cannot provide cost estimates. Also, many DBMS's may provide cost estimates in "cost units" that require conversion into time units. If this conversion factor is unknown or unrelated to physical time, the estimate is useless.

3.2 A Working Cost Model

As can be seen above, many factors can have an effect on the execution time of a global query. Many researchers claim that in the majority of cases, communication costs dominate all other costs [13, 19, 28]. Some of the optimization algorithms presented above rely on assumptions regarding communication costs. A quick experiment (experiment 0) will show

that communication cost outweighs processing costs in the simple case. Table 1 presents the results of a fetch test run to determine how long it takes to fetch all rows from a DBMS and transfer them to *odbcfet.exe* [25] using shared memory or a wired network. The odbcfet.exe utility times how long it takes to fetch data from an ODBC data source. The utility was used in this experiment because it has a very small memory footprint, thus reducing the performance impact to the DBMS when both the utility and DBMS are running on the same machine. This is important to minimize the impact in performance on the DBMS, allowing us to more accurately measure the fetch times of queries in a shared memory environment. As table 1 shows, the time required to complete a query over a wired network is very large compared to the time taken to complete the query over shared memory. Thus, reducing communication cost appears to have a significant impact on query response time, warranting further investigation into the factors affecting it. The queries were run using a hot server cache to ensure that hard disk I/O was not performed. If disk I/O were performed, our measurements would not allow us to compare pure network performance as other factors would be included in the recorded times.

$$Cost = C_{OTHER} + C_{COMM} \tag{7}$$

$$C_{OTHER} = C_{OPT} + C_{EXEC} + C_{SM} + C_{RF} \tag{8}$$

The aim of the rest of this project is to determine what factors most affect the cost of communication. A simplified cost model is used to do so. This model allows the separation of communication cost from all other costs involved in query execution (allowing the study of

Field Number	ANSI Type	Field Number	ANSI Type
1	char(10)	9	real
2	char(50)	10	smallint
3	char(100)	11	time
4	date	12	timestamp
5	decimal	13	varchar(10)
6	double	14	varchar(50)
7	float	15	varchar(100)
8	int		

Table 2: ANSI Data Type for Generated Columns in Four Argument Mode

communication cost to be separated from the study of other costs). The cost model presented in equation (1) is simplified into the working cost model shown in equation (7) with the component C_{OTHER} provided in equation (8). Section 6 will discuss this model in more detail.

4 Developed Applications

Before a discussion of the experiments performed can be given, some applications must be introduced. These applications will allow us to perform our experiments and format our results. This section contains a discussion of the four applications developed: *DBCreate*, a table generation tool developed to build data sets for use during the experiments outlined in section 5; *DBBench*, a benchmarking tool developed to perform the DBMS benchmarking experiments oulined in section 5; *NetBench*, a network benchmarking tool developed to perform the network experiments outlined in section 5; and *ResultParse*, a tool used to parse and organize the data sets obtained by DBBench and NetBench, and used for analysis in section 6.

4.1 DBCreate

DBCreate is a Java utility developed to generate the TYPE_TABLE table data required for the experiments presented in section 5. This utility was developed with the following goals in mind:

1. Generate random data of a specific data type that can be imported into each of the four systems supported by DBBench (see below). Random data is necessary to ensure that data bias is not introduced.

2. Provide a means to generate data of uniform type or data of varying type, depending on the requirements of the experiment the data is to be used for.

Since no utility could be found that would generate data with the specific requirement of complete randomness and without generation in the format of a specific DBMS, the decision was made to develop one from scratch.

The DBCreate utility operates in two modes. The four argument mode generates a table containing 15 data columns of differing data type. Table 2 lists the ANSI SQL data type the individual columns contain. The seven argument mode generates a table with a specified number of rows each containing a specified number of columns of uniform data type. As data is imported into the various systems, it is transformed to conform to each system's implementation of each ANSI type.

DBCreate can be invoked from the command line and requires the specification of either the four mandatory arguments, or the four mandatory and three additional arguments (i.e. either four or seven argument mode):

1. *FIELD_DELIMITER*: The delimiter used to separate field values in the file. Do not use a letter or number here. A pipe, '|', is suggested. This is a mandatory argument.

2. *RECORD_DELIMITER*: The delimiter used to separate records in the file. Do not use a letter or number here. A pipe, '|', is suggested. A line break is automatically added to the end of each line and does not need to be specified in the delimiter. This is a mandatory argument.

3. *NUMBER_OF_RECORDS*: An integer representing the number of records to be created. The application will fill the data file with the number of records specified by this argument. Non-integer values will throw exceptions. This is a mandatory argument.

4. *FILE_NAME*: The fully qualified output path and filename for the generated table. The application will throw an exception if the file cannot be opened for writing. A name such as 'table.tbl' is suggested, placing the file in the same directory as the Java Class. This is a mandatory argument.

5. *DATA_TYPE*: The data type to be used to populate each column in the table. This is an optional argument and no default is assumed. It must be one of:

    - '1': ANSI CHAR datatype
    - '2': ANSI DATE datatype
    - '3': ANSI DECIMAL datatype
    - '4': ANSI DOUBLE datatype
    - '5': ANSI FLOAT datatype

```
>java DBCreate | | 100000 table.tbl
```
-Generate 100 000 rows and store them in a file called 'table.tbl'. Separate the columns and rows with a pipe character, '|'.
```
>java DBCreate | | 50000 char10.tbl 1 100 10
```
-Generate 50 000 rows with 100 columns of character data. Set the field length to 10 characters. Separate each field and each row with a pipe character, '|', and store the output in a file called 'char10.tbl'.
```
>java DBCreate | | 50000 date.tbl 2 100 0
```
-Generate 50 000 rows with 100 columns of date data. Separate each field and each row with a pipe character, '|', and store the output in a file called 'date.tbl'.

Figure 1: Example DBCreate Calls

- '6': ANSI INT datatype
- '7': ANSI REAL datatype
- '8': ANSI SMALLINT datatype
- '9': ANSI TIME datatype
- '10': ANSI TIMESTAMP datatype
- '11': ANSI VARCHAR datatype

6. *NUMBER_OF_COLUMNS*: The number of data columns to include in the data set. Each column will be filled with random data of the type specified in DATA_TYPE above. This is an optional argument and no default is assumed.

7. *DATA_SIZE*: The length of the data field to be created. This argument is only applicable to CHAR and VARCHAR data types. Specify a '0' for all other data types. This is an optional argument and no default is assumed.

All of the above arguments must be specified in the order listed. The user must validate input data as no error checking is performed. DBCreate will notify the user after the creation

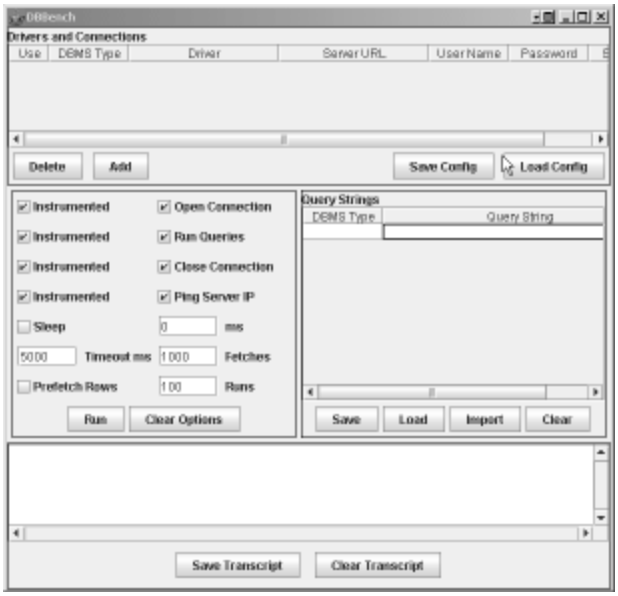

Figure 2: DBBench GUI

of each individual set of 1 000 rows. If an existing file is specified in the FILE_NAME argument, it will be overwritten without warning. DBCreate can be invoked with a call similar to the ones in figure 1.

4.2 DBBench

An application was needed to perform query benchmarking of remote data sources. This application needed to meet the following design goals:

1. Perform benchmarking in a universal manner with four different DBMS's (i.e. no optimized or individualized code for different DBMS's beyond that which is required to control specific parameters such as prefetching, etc.).

2. Allow the user to specify the queries executed, what portions of execution were instrumented, the status of prefetching, the servers contacted, etc.

3. Be deployable on any platform without the need for redevelopment.

4. Allow control via command line scripting for automated testing to be performed.

5. Support connection to and querying of several different DBMS's.

Since no application could be found that met all the design goals, the decision was made to develop one. Java was chosen as the language to allow universal deployment while using the standard JDBC driver calls (thus supporting several DBMS's). DBBench is a Java Swing application developed to record performance measurements for queries, etc. executed on remote data sources. DBBench has many features:

- *Instrument connection open and close, query open, row fetch and server ping times*: DBBench can record the time taken to open a connection to a server, close a connection to a server, submit a query and open a cursor, fetch a configurable number of rows, and ping an IP address.

- *Configurable options*: DBBench can be configured to instrument any number of row fetches and any number of repetitions for the posed query set. Prefetching can also be enabled or disabled.

- *Different Servers*: DBBench can instrument iAnywhere Solutions Adaptive Server Anywhere, Microsoft SQL Server 2000, Oracle Database 10g, IBM DB2 8.1, and Sybase Adaptive Server Enterprise using publically available JDBC drivers.

DBBench can run and instrument any number of queries on any of the five supported servers. Results are recorded in text files that are human-readable and parseable by ResultParse (below). DBBench was created with an easy to use GUI that provides feedback regarding

the status of a query set upon its completion. See figure 2 for a look at the DBBench GUI. DBBench can also be invoked from the command line using a rigid parameter structure as outlined below:

1. *NUMBER_OF_FETCHES*: The number of fetches to perform for this test. Non-integer arguments will throw an exception. DBBench will perform up to the specified number of fetches when running a query. DBBench will stop prematurely if the record set contains less than the specified number of records.

2. *NUMBER_OF_RUNS*: The number of times to repeat each query. Non-integer arguments will throw an exception. DBBench will perform each query this number of times.

3. *PING_TIMEOUT*: The amount of time (in milliseconds) to wait before aborting a ping timing. Non-integer arguments will throw an exception.

4. *SLEEP_TIME*: The amount of time (in milliseconds) to pause between successive executions. Non-integer arguments will throw an exception. DBBench will pause for this amount of time before starting the next run or the next query within a run.

5. *INSTRUMENT_OPEN*: '0' or '1'. A value of '1' tells DBBench to record the amount of time it takes to open a connection to the server.

6. *OPEN_CONNECTION*: '0' or '1'. A value of '1' tells DBBench to open a connection to the server.

7. *INSTRUMENT_QUERY*: '0' or '1'. A value of '1' tells DBBench to record the amount of time it takes to open the result set, fetch each row, and the total amount of time

required to open the result set and request each row (i.e. the total query cost, not the fetch or query execution costs individually).

8. *RUN_QUERY*: '0' or '1'. A value of '1' tells DBBench to run the queries specified in the query list.

9. *INSTRUMENT_CLOSE*: '0' or '1'. A value of '1' tells DBBench to record the amount of time it takes to close the connection to the server.

10. *CLOSE_CONNECTION*: '0' or '1'. A value of '1' tells DBBench to close the connection to the server.

11. *INSTRUMENT_PING*: '0' or '1'. A value of '1' tells DBBench to record the ping time to the server's IP address.

12. *PING_ADDRESS*: '0' or '1'. A value of '1' tells DBBench to ping the IP address of the server as specified in the drivers table.

13. *SLEEP*: '0' or '1'. A value of '1' tells DBBench to pause between execution of each query and each run. The pause duration is specified in the SLEEP_TIME argument.

14. *PREFETCH*: '0' or '1'. A value of '1' tells DBBench to execute queries with prefetch enabled.

15. *QUERY_FILENAME*: This is the fully qualified path to a file of queries to be imported and executed. See below for the import file specification. Do not use spaces in this pathname.

```
>java Main 200000 1 1000 1000 1 1 1 1 1 1 0 0 0 1
asa-tpch.sql config-asa-1.dbt 1
```
-Execute DBBench to perform 1 run with 200 000 fetches. Set the sleep time and ping timeout to 1 second. Instrument and perform a connection open, query execution and connection close. Do not ping the server addres or sleep between executions. Turn prefetching on for the queries stored in the 'asa-tpch.sql' file. Use the driver configuration stored in the 'config-asa-1.dbt' file and run the queries.
```
>java Main 1 100 500 500 0 1 0 1 0 1 1 1 1 0
asa-tpch-max.sql config-asa-2.dbt 1
```
-Execute DBBench to perform 100 runs with 1 fetch per run. Set the sleep and ping timeout to 0.5 seconds. Do not instrument connection open, query execution or connection close: however, perform all three operations. Instrument and ping the server address. Sleep between query executions, but do not use prefetching. Execute the queries stored in the 'asa-tpch-max.sql' file. Use the driver configuration stored in the 'config-asa-2.dbt' file and run the queries.

Figure 3: Sample DBBench Command Line Executions

16. *CONFIG_FILENAME*: This is the fully qualified path to a file of driver configurations that are to be used to execute the imported queries. Drivers must be specified in DBBench and saved to a configuration file first. This file may then be used for query execution. Do not use spaces in this pathname.

17. *RUN*: '0' or '1'. A value of '1' tells DBBench to execute the queries imported using the specified drivers and execution parameters. A value of '0' tells DBBench to set the configuration parameters provided in the command line call, but not to execute the queries. This feature allows the user to check that a configuration specified in a command line call to DBBench is correct before execution of the queries begins.

Each argument is required and must be specified exactly as outlined. Invalid arguments will throw exceptions. DBBench will automatically quit after query execution has completed.

The transcript generated during execution will be written to the file 'transcript.dbt' which is human-readable using an editor such as Microsoft WordPad. See figure 3 for some sample DBBench command line executions.

As mentioned above, queries must be specified in an import file. The import file must contain the following arguments (on separate lines) before any queries are specified. Comments (see below) may appear anywhere in the file but must be on their own line:

1. 'dbms': This argument speficies the DBMS type to be benchmarked. It must be one of 'asa' (iAnywhere Adaptive Server Anywhere 9.0), 'mss' (Microsoft SQL Server 2000), 'db2' (IBM DB2 8.1), 'oracle' (Oracle Database 10g) or 'ase' (Sybase Adaptive Server Enterprise 12.5).

2. 'delim': This argument specifies the delimiter used to separate queries within the import file. A ';' is suggested in order to avoid using character strings that may appear in the query text (such as the word 'go').

3. '–': This argument may appear multiple times throughout the file and is used to denote that the current line is a comment. Comments must appear on a line by themselves and not, for example, at the end of a line containing a query.

It is suggested that arguments occupy the first few lines of the import file. See figure 4 for a sample import file.

Instrumentation in DBBench is quite simple. Figure 5 shows the instrumentation surrounding several portions of query submission and fetch code. The recorder starts immediately prior to issuing the related request and stops immediately after the request has completed.

```
–REQUIRED ARGUMENTS
dbms=asa
delim=;

–PERFORM SELECT QUERY FIRST
SELECT COL1, COL2
FROM TEST_TABLE
WHERE COL1 > 1000;

–PERFORM JOIN QUERY NEXT
SELECT T.COL1, T.COL2
FROM T, R
WHERE T.COL1 = R.COL1;
```

Figure 4: Sample Query Import File

The methods used to start and stop the recorder, and calculate the total time are presented in figure 6. The recorder itself uses nanosecond precision and accuracy obtained with the `System.nanoTime()` method in JDK 1.5.0 [18]. As a result, extremely accurate time measurements can be taken.

4.3 NetBench

To understand how well different DBMS's use the network, an application was required to perform benchmarking on the actual network connection between the servers and the client machine. This utility needed to meet the following design goals:

1. A Java implementation was necessary to facilitate comparison of the times obtained with DBBench (also developed in Java).

2. The implementation should impose as little overhead as possible during communication between the client and server portions. This is necessary to ensure that accurate

```
public String runInstrumentedQuery(String queryString, int numberToFetch,
                                   boolean prefetch) {
        //This if statement disables prefetching for system 3
        if(dbmsType.compareTo("system_3") == 0 && !prefetch)
                stmt = connection.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                   ResultSet.CONCUR_UPDATABLE);
        else
                stmt = connection.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                   ResultSet.CONCUR_READ_ONLY);
        //This statement disables prefetching for system 2 and system 4
        if(!prefetch)
                stmt.setFetchSize(1);
        //system 1 requires this line when performing max queries
        if(prefetch && dbmsType.compareTo("system_1") == 0 &&
                                   queryString.contains("max"))
                stmt.setFetchSize(1);
                                   :
        Thread.sleep(0);
        totalRecorder.start();
        queryRecorder.start();
        ResultSet rs = stmt.executeQuery(queryString);
        queryRecorder.stop();
        queryRecorder.reset();
        //This if statement disables prefetching for system 2 and system 4
        //system 1 prefetch is done in connection string
        if(dbmsType.compareTo("system_1") != 0 && !prefetch)
                rs.setFetchSize(1);
        boolean keepGoing = true;
        for(int i = 0; i < numberToFetch & keepGoing; i++) {
                Thread.sleep(0);
                fetchRecorder.start();
                keepGoing = rs.next();
                fetchRecorder.stop();
                fetchRecorder.reset();}
        totalRecorder.stop();
                                   :
}
```

Figure 5: DBBench code instrumentation

```
public boolean start() {
        startTime = System.nanoTime();
        return true;
}
                        ⋮
public boolean stop() {
        endTime = System.nanoTime();
        return true;
}
                        ⋮
public boolean reset() {
                        ⋮
        filePointer.print(trialNumber+"\t");
        filePointer.print(startTime+"\t");
        filePointer.print(endTime+"\t");
        filePointer.print(endTime-startTime+"\r\n");

        endTime = 0;
        startTime = 0;
        trialNumber++;
                        ⋮
}
```

Figure 6: Recorder methods for starting and stopping time

measurements of actual maximum network throughput are taken.

Since no utility could be located for free that met the design goals, the decision was made to develop one. NetBench is a Java utility developed to benchmark network performance between two machines connected via a network. It was intentionally designed to use no GUI and minimal code so that very precise timings can be taken with minimal overhead incurred. Data for each test is generated in the same format as the data generated by DBBench, and is thus human-readable and parseable by ResultParse (below). NetBench has a rigid argument structure containing the following arguments:

1. *HOST*: This is the IP address of the host. For SEND_MODE, it is the destination address that the client will connect to. For LISTEN_MODE, it is the address of the server instance of NetBench.

2. *PORT*: This is the port number of the host. For SEND_MODE, it is the destination port that the client will connect to. For LISTEN_MODE, it is the port number the NetBench server instance will listen on. Non-integer values will throw an exception.

3. *MODE*: This is the mode that this instance of NetBench will operate under. Non-integer values will throw an exception. The specified value must be one of the following:

    - *LISTEN_MODE*: '0'. This value tells NetBench to operate as a server: accepting incoming data from clients.
    - *SEND_MODE*: '1'. This value tells NetBench to operate as a client. Clients send data to servers and instrument the time it takes to send the data accross the network.

4. *BYTES*: This is the size of a data packet to be sent (in bytes). When operating in SEND_MODE, this is the size of a data packet that will be sent PACKETS times. When operating in LISTEN_MODE, this is the size of an acknowledgement packet that is sent for every packet received. Non-integer values will throw an exception.

5. *PACKETS*: This argument specifies the number of packets to be sent. Non-integer values will throw an exception. Specify a value of '0' for instances running in LISTEN_MODE.

6. *RUNS*: This argument specifies the number of runs to perform. When operating in LISTEN_MODE, this is the number of runs that NetBench will listen for before exiting (as per the KILL argument below). When operating in SEND_MODE, this is the number of runs that NetBench will perform with the given parameters. Non-integer values will throw an exception.

7. *KILL*: When running in LISTEN_MODE, this argument tells NetBench to exit after receiving data from the number of runs specified in RUNS. When in SEND_MODE, NetBench will automatically exit when it has completed the specified number of runs. The argument must be one of:

    - *'true'*: Exit when the number of runs has been received.
    - *'false'*: Do not exit. Receive an unlimited number of runs.

    Either value may be specified when running in SEND_MODE.

NetBench contains two components: a listener (server) and a sender (client). Only one component is active for each instance of NetBench (i.e. there is no active client when NetBench is

`>java NetBench 10.0.0.2 65001 0 8 0 10 true` -Launch this instance of NetBench as a listener awaiting incoming connections on port 65 001. Exit after 10 runs have been received.
`>java NetBench 10.0.0.2 65001 1 10 50000 10 false` -Launch this instance of NetBench as a sender. Send 50 000 packets containing 10 bytes of data each to the listener at address 10.0.0.2 on port 65 001. Repeat the run 10 times.

Figure 7: Example NetBench Calls

invoked as a listener). Two versions of NetBench have been deployed: one version will compile with JDK 1.4.2 [17]; the second version compiles with JDK 1.5.0 [18]. The latter version was developed to make use of the nanosecond precision and accuracy of the `System.nanoTime()` function for timing measurement. The former version was developed to be deployed without requiring additional software (JDK 1.5.0) installations. It is recommended that NetBench senders be run on machines with JDK 1.5.0 installed to achieve the most accurate measurements possible. Since NetBench listeners do not make any time measurements, they may be installed on a machine with either JDK version 1.4.2 or 1.5.0.

Output from NetBench tests will be written to the file 'send.txt' in the same format as fetch results from DBBench. Thus, they may be parsed by ResultParse using the SPLIT operation with FETCH suboperation (see below). See figure 7 for sample calls to NetBench.

4.4 ResultParse

Although DBBench and NetBench provide results in a human-readable format, when automated testing is performed the format of the data may require alteration in order to be useful for analysis. As such, an application was needed to transform the results obtained in section 5 into a more usable format. Since the data format generated by DBBench and NetBench is

proprietary, no other transformation tool could be useful. Thus, one was developed.

ResultParse is a Java utility developed to parse the fetch and query result files created by DBBench and NetBench. Data for each specific test is written to its own file. ResultParse requires a very rigid argument specification with the following arguments:

1. *FILE_NAME*: This is the fully qualified path to the input file.

2. *OPERATION*: This is the operation to be performed by ResultParse and must be one of:

   - *'SPLIT'*: Split one input file into multiple output files. This will take a file containing results for several different query and fetch tests and save the results for each test into its own file.
   - *'JOIN'*: Combine multiple input files into one output file. This is useful for looking at many different output files together in a spreadsheet for comparison.
   - *'STAT'*: Add statistics generation footer for use in Excel. This is useful for automatically calculating the max, min, average, etc. of a result file.
   - *'ALTER'*: Alter the results contained in a result file. This allows the scale of the recorder to be changed after recording has taken place. NOTE: Digits are truncated, not rounded.

3. *SUBOPERATION*: This is the suboperation to be performed and must be one of:

   - *'QUERY'*: The input file(s) are from a query test, not a fetch test.
   - *'FETCH'*: The input file(s) are from a fetch test, not a query test.

- *'MERGE'*: The input files are merged and the full result line for each file is left intact (i.e. the trial number, recorder start time and recorder end time are not removed from the line before writing it to the merged file).
- *'STRIP'*: The input files are merged and only the recorder time is printed for each file (i.e. the trial number, recorder start time and recorder end time are removed from the line before writing it to the merged file).
- *'NOOP'*: No suboperation to be performed.

QUERY and FETCH are only applicable to the SPLIT operation while MERGE and STRIP are only applicable to the JOIN operation. NOOP is a required suboperation for STAT and ALTER operations.

4. *MAGNITUDE*: This argument specifies the magnitude for each of the operations as defined below:
   - *'SPLIT'*: The number of output files that should be created.
   - *'JOIN'*: The number of input files that should be read.
   - *'STAT'*: 0. Number is not used but must still be present.
   - *'ALTER'*: The number of digits to be truncated.
5. *OUTPUT_PATH*: The location where output files should be placed. This must be a fully qualified path *without* a file name.
6. *FILE_LIST - or - SECOND_SET*: This argument will either begin a file list for use in the JOIN operation, or specify the SECOND_SET argument.

- *FILE_LIST*: This argument and every argument after it are optional and specify one of the input files required for the JOIN operation. The number of files specified in MAGNITUDE must be entered as arguments. Separate each fully qualified file path (with file name) with a space.

- *SECOND_SET*: This argument is optional and can be specified for the SPLIT operation with FETCH suboperation. By default, ResultParse will split the second set of fetch results listed in the result file. For example, if the fetch test has been repeated more than once, the second set of results will be used and all other results will be omitted from the split files. The default option is useful when performing an operation multiple times to ensure the results are parsed from a query run with a hot cache. Setting SECOND_SET to FALSE will force ResultParse to use the first set of results listed in the result file. This is useful if a test is being repeated.

Any missing arguments from 1 to 6 will cause an exception to be thrown. File paths can not include spaces. Specifying an incorrect argument may generate unwanted results. A non-integer MAGNITUDE argument will throw an exception. Output files will be overwritten without warning if they already exist. Output file names for SPLIT operations will be chosen by ResultParse according to the file number and data table specified in the query. Some example calls to ResultParse are given in figure 8.

`>java ResultParse query.txt SPLIT QUERY 15 214/query/` -Split the query results stored in query.txt into 15 output files and store the result files in the subfolder 'query' of the folder '214'.
`>java ResultParse fetch.txt SPLIT FETCH 8 114/fetch/` -Split the fetch results stored in fetch.text into 8 output files and store the result files in the subfolder 'fetch' of the folder '114'.
`>java ResultParse ping.txt ALTER NOOP 3 114/alter/pingalter.txt` -Alter the file ping.txt by truncating 3 digits from each result value and place the result in a file called 'pingalter.txt' in the subfolder 'alter' of the folder '114'.
`>java ResultParse alter/ping.txt STAT NOOP 0 114/` -Add statistics calculators to the file 'ping.txt' and store the result file in the folder '114'.
`>java ResultParse merge-open.txt JOIN STRIP 3 1xinstrument/ 114/open.txt 124/open.txt 134/open.txt` -Join the 'open.txt' files in folders '114', '124' and '134'. Strip off the extraneous information and store the result file in the folder '1xinstrument'.

Figure 8: Example ResultParse Calls

5 Experimental Setup

Now that we have outlined the applications we will use to run our experiments, we can provide an outline of their setup and execution. This section provides an overview of the hardware and software setup used to obtain the results in section 6. A description of the procedure used to gather results is also presented.

5.1 Hardware Setup

Experiments were performed using three different machines with the hardware specifications shown in table 3. The four hard drives in the client machine were used as follows: Drive C was used for system and application software; Drive D was used to hold queries and data sets for

Brand	Dell	Dell	Dell
Model	Precision 410	Precision 410	OptiPlex GX150
RAM	128 MB SDRAM	192 MB SDRAM	256 MB SDRAM
HD	SCSI: 4x8 GB	SCSI: 2x8 GB	IDE: 1x6 GB; 1x15 GB;
Network	10/100BaseT	10/100BaseT	10/100BaseT
Processor	400 MHz Pentium II	450 MHz Pentium II	800 MHz Pentium III
	Client	Server 1	Server 2

Table 3: Machine Hardware Specification for Experiments

Brand	SMC	Linksys
Model	EtherEZ 3605T	EZXS88W
Speed	10BaseT	10/100BaseT
	Hub 1	Hub 2

Table 4: Hub Hardware Specification for Experiments

experiments; Drive E was used to hold extraneous files; Drive F was used to hold the results from experiments. The two hard drives in the server machines were used as follows: Drive C was used for system and DBMS software; Drive D was used to hold data files for each of the four DBMS's (i.e. log files, data tables, etc.).

The client and server were connected using Hubs 1 and 2 from table 4. An uplink connection providing access to the network and a DHCP server was used to transfer results as well as test scripts to and from the test machines. The uplink cable was disconnected while tests were run to ensure network traffic did not affect results.

5.2 Software Setup

The machines discussed above were configured with the software components outlined in table 5. JBuilder was used as a development environment for DBCreate, DBBench, NetBench and ResultParse. See section 4 for more information about these applications. Ethereal is a packet

OS	Windows XP Professional SP2	Windows XP Professional SP2
Applications	Borland JBuilder Ethereal	
DBMS's		System 1 System 2 System 3 System 4
JDK	1.5.0	1.4.2
	Client	Server 1 and 2

Table 5: Software Specification for Experiments

sniffing application that was used to confirm the status of prefetching, as well as retrieve the sizes of packets sent by the various systems. JDK 1.5 [18] was used on the client machine to allow DBBench and NetBench to use nanosecond precision and accuracy. Due to the licensing agreement in place for the use of different DBMS's, it is not possible to publish product names along with performance results. As such, the DBMS's have been renamed to System 1 - System 4. JDK 1.4.2 [17] was used on the server machines as support for the applications and utilities that compose System 1, 3 and 4.

5.3 Experimental Procedure

Two experiments were performed to obtain the results used for analysis in section 6. Experiment 1 was performed to determine the communication time at the actual maximum data rate between the client and server machines. This time is important to determine if the problem of communication cost modeling can be solved through measurements of actual network throughput (not theoretical maximums). Experiment 2 was performed to determine how row size, data type, link speed, prefetching status, network packet size, server CPU speed and DBMS affect communication cost.

5.3.1 Experiment 1

Experiment 1 was performed to determine the communication time at the actual maximum data rate between the client and server machines (i.e. using raw sockets without any DBMS overhead involved). The experiment proceeded as follows:

1. Connect server 1 and the client machine using hub 1. Do not connect the hub to the main network so that network traffic does not impact results.
2. On server 1, compile and launch NetBench. Configure it to listen on port 65 001.
3. On the client, compile and launch NetBench using configuration 1 from appendix A.
4. Repeat step 3 with configuration 2 - 26 instead of configuration 1.
5. Repeat steps 1 - 4 using server 2 instead of server 1.
6. Repeat steps 1 - 5 using hub 2 instead of hub 1.

NetBench was configured to send 50 000 data blocks during the experiment so that times measured can be compared to the times measured during experiment 2 (in which each data table will contain 50 000 rows). The experiment was repeated 30 times to generate a statistically significant data set with relative error below 5%.

5.3.2 Experiment 2

Experiment 2 was performed to determine how row size, data type, link speed, prefetching status, network packet size, server CPU speed and DBMS affect communication cost. The experiment proceeded as follows:

System	Packet Size 1	Packet Size 2
System 1	1460	2920
System 2	4096	2048
System 3	32767	16384
System 4	2048	1024

Table 6: Packet Sizes for Experiment 2

1. Connect server 1 and the client machine using hub 1. Do not connect the hub to the main network, ensuring that the network traffic does not impact results.

2. On server 1, load system 1 and generate the data tables using each schema from appendix C.

3. On server 1, load data into the data tables created in step 2. Shut down system 1.

4. Repeat steps 2 and 3 for systems 2 - 4.

5. Repeat steps 1 - 4 for server 2.

6. On server 1, set the packet size for system 1 to packet size 1 from table 6.

7. Defragment all hard drives on server 1 and the client machine.

8. Restart server 1 and the client machine.

9. On server 1, load system 1.

10. On the client, launch DBBench and disable prefetching.

11. Lock server 1 to ensure that incurred operating system overhead is minimal. This allows system 1 to maximize its performance.

12. In DBBench, execute the queries listed in appendix B and record the total time taken to execute the query and fetch the entire result set.

13. Repeat steps 6 - 12 for systems 2 - 4.

14. Repeat steps 6 - 13 using server 2 instead of server 1.

15. Repeat steps 6 - 14 using hub 2 instead of hub 1.

16. Repeat steps 6 - 15 with prefetching enabled in DBBench.

17. Repeat steps 6 - 16 using packet size 2 instead of packet size 1 for each of systems 1 - 4.

DBBench was configured to fetch 50 000 rows during the actual row tests. This forced the system to scan through the entire data table in memory. Actual row tests were repeated 30 times. During the column maximum tests, DBBench was configured to fetch the result row 1000 times. Thus, a statistically significant set of results with relative error below 5% was obtained for use in section 6. See appendix B for query definitions and appendix C for schema definitions.

6 Experimental Results

Figure 9 presents a simplified view of the query instrumentation performed in DBBench. Time T_1 is measured, the result set is opened, each result row is fetched, the result set is closed, and time T_2 is measured. The goal is to isolate the communication cost from the simplified cost model. As was discussed in the experimental procedure, the query execution times for actual rows and the calculation of the maximum value in a column were instrumented using

```
T1
resultSet.open(<query_string>);
while(<more_records>)
        resultSet.next();
resultSet.close();
T2
```

Figure 9: Simplified View of DBBench Query Instrumentation

sample queries posed over tables containing 100 columns of the same data type. Each data table contained columns of a different data type. Equation (7) presents the simplified cost model that will be assumed for analysis purposes. In order to isolate communication cost it is necessary to remove the cost, C_{OTHER}, of query execution from the cost model. This step can be completed in several different ways. One way is to subtract the time required to fetch a column maximum, C_{MAX}, from the time required to fetch the actual row data, C_{ROW}, as seen in equation (9). These queries were chosen specifically for their simplicity. "SELECT *" queries force the DBMS to load and touch each row in a given table. "SELECT MAX(...)" queries require the same amount of overhead (i.e. load and scan through each row in the given table) but only return one result row. As such, the time it takes to perform a SELECT MAX(...) query can be subtracted from the time it takes to perform a SELECT * query to remove the execution overhead from recorded timings, leaving the communication cost for analysis. Three basic assumptions must be made:

1. *The data tables are in memory*: With the data tables in memory during query execution, no disk accesses are performed and disk I/O is not a factor in recorded times.

2. *Negligible cost to transfer the maximum value*: The communication cost of transmitting the maximum column value result set is negligible.

t_1
resultSetA.open(SELECT * FROM TABLE);
 	
while(<more_records_in_A>)
resultSetA.next();
resultSetA.close();
t_2
	m_1
	resultSetB.open(SELECT MAX(COLUMN)
	FROM TABLE);
	while(<more_records_in_B>)
	resultSetB.next();
	resultSetB.close();
	m_2

Actual Row Fetch	Column Maximum Fetch

Figure 10: Simplified View of Query Instrumentation for Actual Row Fetch and Column Maximum Fetch

3. *Negligible loop evaluation cost*: The cost of evaluating the loop condition during query execution is negligible.

Instrumentation for these queries can be seen in figure 10.

$$C_{COMM} = C_{ROW} - C_{MAX} \tag{9}$$

With the instrumentation in figure 10, equations (10) and (11) can be built.

$$C_{ROW} = t_2 - t_1 \tag{10}$$

$$C_{MAX} = m_2 - m_1 \tag{11}$$

By substituting equations (10) and (11) into equation (9), we are left with the final communication cost calculation presented in equation (12).

$$C_{COMM} = (t_2 - t_1) - (m_2 - m_1) \tag{12}$$

Source	Average Link Speed Difference (% Reduction)	Average Prefetch Difference (% Reduction)	Average Packet Size Difference (% Reduction)	Average CPU Speed Difference (% Reduction)
System 1	23.79	84.14	2.30	11.29
System 2	12.34	87.90	1.08	6.37
System 3	36.37	79.66	0.76	10.69
System 4	20.61	75.82	-2.52	6.56
NetBench	48.90	99.58	1.02	12.54

Table 7: Average Differences for Variables

This final equation allows us to calculate communication cost from the timings gathered during the experiments outlined above. Other methods of calculating the network time may also exist. This method, however, allows us to gather a statistically significant data set using very simple queries. Using these simple queries, we can also say with reasonable assurance that each DBMS will do a similar amount of work, making comparisons between systems possible.

6.1 Results

This section presents an overview of observations obtained from the data gathered in section 5. Graphs describing the data can be found in appendix D. Two types of graphs are provided: scatter plots show the overall trend of fetch times on a per row basis as row size is increased, and bar charts show the overall trend of fetch times on a per row basis as data type is varied. Table 7 shows the average difference between per row fetch times based on a variance of several criteria. The difference is given as a percentage reduction in fetch time as the variable (i.e. link speed, prefetch status, packet size, CPU speed) is increased (or enabled, for prefetching).

1. *DBMS*: As can be seen from table 8, 9, 10 and 11, there is a large difference between DBMS software in fetch times per row (note that the minimum, median and maximum

Packet Size (bytes)	CPU Speed (MHz)	Link Speed MBit/Sec	Prefetch Status (1=On;0=Off)	Minimum (ms)	Median (ms)	Maximum (ms)
1460	450	10	0	0.8028	1.0730	2.6962
1460	450	10	1	0.1032	0.2896	1.5936
1460	450	100	0	0.6974	0.8380	1.6250
1460	450	100	1	0.0940	0.1750	0.6600
1460	800	10	0	0.6674	0.9222	2.3694
1460	800	10	1	0.1102	0.2090	1.4152
1460	800	100	0	0.5564	0.6788	1.3080
1460	800	100	1	0.1086	0.1748	0.4636
2920	450	10	0	0.8104	1.0776	2.8546
2920	450	10	1	0.1106	0.2136	1.4242
2920	450	100	0	0.6946	0.8330	1.6612
2920	450	100	1	0.1028	0.1766	0.7778
2920	800	10	0	0.6718	0.9302	2.5368
2920	800	10	1	0.1138	0.2190	1.4206
2920	800	100	0	0.5664	0.6832	1.3324
2920	800	100	1	0.1084	0.1762	0.6820

Table 8: System 1 Data Ranges

Packet Size (bytes)	CPU Speed (MHz)	Link Speed MBit/Sec	Prefetch Status (1=On;0=Off)	Minimum (ms)	Median (ms)	Maximum (ms)
4096	450	10	0	1.2644	1.5728	4.3286
4096	450	10	1	0.1270	0.2362	2.2414
4096	450	100	0	1.0822	1.2686	3.5740
4096	450	100	1	0.1272	0.2280	2.1466
4096	800	10	0	1.0744	1.3778	4.1890
4096	800	10	1	0.1298	0.2356	2.1958
4096	800	100	0	0.8876	1.0712	3.5816
4096	800	100	1	0.1292	0.2294	2.1502
2048	450	10	0	1.2804	1.5796	4.4414
2048	450	10	1	0.1280	0.2376	2.2016
2048	450	100	0	1.0868	1.2694	3.6246
2048	450	100	1	0.1292	0.2300	2.2026
2048	800	10	0	1.0810	1.4282	4.2980
2048	800	10	1	0.1300	0.2396	2.2390
2048	800	100	0	0.9000	1.0904	3.5660
2048	800	100	1	0.1300	0.2288	2.1538

Table 9: System 2 Data Ranges

Packet Size (bytes)	CPU Speed (MHz)	Link Speed MBit/Sec	Prefetch Status (1=On;0=Off)	Minimum (ms)	Median (ms)	Maximum (ms)
32767	450	10	0	0.7996	1.0190	3.2946
32767	450	10	1	0.1406	0.3324	2.7404
32767	450	100	0	0.6398	0.7484	1.7610
32767	450	100	1	0.1350	0.1860	0.6080
32767	800	10	0	0.6896	0.9080	3.1708
32767	800	10	1	0.1380	0.3218	2.6934
32767	800	100	0	0.5350	0.6358	1.3108
32767	800	100	1	0.1288	0.1666	0.5842
16384	450	10	0	0.8330	1.0318	3.3496
16384	450	10	1	0.1396	0.3272	2.7392
16384	450	100	0	0.6572	0.7642	1.6982
16384	450	100	1	0.1372	0.1958	0.5960
16384	800	10	0	0.7128	0.9040	3.1630
16384	800	10	1	0.1326	0.3180	2.6920
16384	800	100	0	0.5464	0.6562	1.3240
16384	800	100	1	0.1294	0.1632	0.5552

Table 10: System 3 Data Ranges

Packet Size (bytes)	CPU Speed (MHz)	Link Speed MBit/Sec	Prefetch Status (1=On;0=Off)	Minimum (ms)	Median (ms)	Maximum (ms)
2048	450	10	0	0.9986	1.2402	2.7398
2048	450	10	1	0.2240	0.4406	1.3236
2048	450	100	0	0.8400	1.0102	1.8326
2048	450	100	1	0.1938	0.3390	0.7854
2048	800	10	0	0.8274	1.1098	2.4562
2048	800	10	1	0.2076	0.4422	1.3398
2048	800	100	0	0.6868	0.8158	1.9280
2048	800	100	1	0.1798	0.3750	1.7312
1024	450	10	0	0.9864	1.2308	2.6450
1024	450	10	1	0.2176	0.4438	1.3218
1024	450	100	0	0.8370	1.0104	1.8168
1024	450	100	1	0.1934	0.3326	0.8154
1024	800	10	0	0.8078	1.0112	2.4198
1024	800	10	1	0.1976	0.4352	1.3600
1024	800	100	0	0.6592	0.8240	1.5718
1024	800	100	1	0.1784	0.3476	0.7874

Table 11: System 4 Data Ranges

times shown are per row fetch time). The median was chosen instead of the mean as outliers have a large impact on averages. Looking at the difference in median values by source illustrates that each DBMS has different methods of, and overheads associated with, sending data over the network. Thus, one must experiment with each system to be modeled to generate a model that represents all the required DBMS's.

2. *Link Speed*: As can be seen from table 7, as well as from a comparison of the scatter plots in appendix D, an increase in link speed has a large impact on fetch time per row. Systems saw a decrease in median fetch time of 12-36% per row. Further, the slope of the graphs appears to decrease as link speed increases; meaning that an increase in link speed more greatly affects larger data sizes.

3. *Data Size*: An analysis of the scatter plots in appendix D shows that fetch time per row increases as data size increases. This is seen in the upward slope of the scatter plots.

4. *Data Type*: An analysis of the bar charts in appendix D shows that a small difference in fetch time per row can be seen between data types. This is visible by noting the slight differences between each ANSI data type and the fetch times of the CHAR or VARCHAR data of the same number of bytes. It should be noted that each system, with the exception of system 4, appears to have significantly greater overhead associated with accessing date, time and timestamp data as opposed to numeric and character data (which both experience fairly consistent fetch times).

5. *Prefetch Status*: The scatter plots in appendix D as well as the percentage decrease due to prefetch status in table 7 show that prefetching can have a large impact on the fetch

time per row. Each system noted a decrease in median fetch time of 75-84% per row. Thus, prefetching has a very large impact on communication cost.

6. *Packet Size*: An increase in network packet size appears to have a negligible impact on median fetch time per row; this can be seen in the small percentage change due to packet size in table 7. Each system noted a change of no more than 3% in median fetch time, and the change could easily be attributed to data error.

7. *Server CPU Speed*: As can be seen in table 7 and the scatter plots in appendix D, server CPU speed does have a slight impact on median fetch times. Each system saw a decrease in fetch time of 6-11%. Thus, while not as significant an impact as prefetch status or link speed, server CPU speed does impact network time.

As noted in section 3, and proven through experimentation in section 5 as well as result analysis above, several factors have an impact on the amount of time a system must spend transmitting data over the network.

6.2 Other Notes

Some other information can be extracted from the analysis of the collected data.

6.2.1 Dominant Factors

As seen above, some factors affect the fetch time per row more than others. The dominant factors seen above appear to be DBMS, link speed, prefetch status and data size. The dominance of each factor can be seen in the large percentage reduction in median fetch time in table 7, as well as the shape of the scatter plots found in appendix D.

Source	Average Relative Error (%)
System 1	0.0274
System 2	0.2026
System 3	0.5756
System 4	0.5043
NetBench	0.0023

Table 12: Average Relative Error for Fetch Times

Other factors, such as data type, packet size and server CPU speed, while having some impact, do not have nearly as large an impact as the four dominant factors noted above.

6.2.2 Consistency of the Systems

Table 12 shows the average relative error in the fetch times obtained for each system. The low relative error (below 1% for each system) demonstrates that each system is consistent in its network transmission overhead. So, although other factors such as network traffic may affect the consistency of fetch times, the DBMS overhead should remain relatively constant.

It should be noted that although each DBMS has a very low relative error, tests performed using NetBench had a relative error 2 - 2.5 orders of magnitude lower than the DBMS's. So, while they are consistent in their usage of the network, they can do better, and some variance in times should be expected in a real world deployment.

6.2.3 Efficiency of Link Usage

Each DBMS adds some overhead to the network transmission time. Table 13 is provided for comparison to table 8-11. Table 14 shows the increase in fetch time per row over the time taken to fetch data through the network using NetBench. Overhead such as character and value

Packet Size (bytes)	CPU Speed (MHz)	Link Speed MBit/Sec	Prefetch Status (1=On;0=Off)	Minimum (ms)	Median (ms)	Maximum (ms)
1460	450	10	0	0.4726	0.5854	1.3166
1460	450	10	1	0.0024	0.1610	1.1668
1460	450	100	0	0.3702	0.3972	0.5700
1460	450	100	1	0.0010	0.0632	0.4530
1460	800	10	0	0.3940	0.5071	1.2196
1460	800	10	1	0.0024	0.1567	1.1092
1460	800	100	0	0.2912	0.3176	0.4822
1460	800	100	1	0.0008	0.0543	0.3892

Table 13: NetBench Data Ranges

Source	Average Time Increase (% Increase)
System 1	173.04
System 2	239.14
System 3	177.44
System 4	261.86

Table 14: Average Increase in Fetch Time Over NetBench

translation, packet formation, etc. are possible causes for the staggering increases in times. NetBench is using the network as fast as possible and sending only raw byte data; while the DMBS's may require some data translations be performed before forming and transmitting a packet of data. 173-261% of the time required to send data between the NetBench sender and listener is required to send data of the same size between DBBench and the various DBMS's. This is an astonishing increase in fetch time, showing that modeling network transmission time for DBMS's cannot be done by modeling communication between two network applications (i.e. something special happens inside a DBMS when sending data that is unique to it and must be measured and modelled using the DBMS itself).

6.3 Recommendations

After careful analysis of the data presented above, the following recommendations can be made to improve the global query optimization strategy in ASA:

1. *Minimize Data Size*: Data size has a large impact on fetch time. Thus, data should be reduced at the remote data sources to the greatest extent possible. Further, an attempt should be made to remove expensive data types during the reduction phase. Data types such as timestamps, dates, etc. that take significantly longer to fetch should be removed from the result set whenever possible.

2. *Use Prefetching*: Enable prefetching whenever a result set larger than one packet is expected. The most appropriate time to use prefetching is when a large data set is to be retrieved. Using prefetching on small data sets may incur additional setup overhead,

actually increasing average fetch times for data sets with a small number of rows (i.e. less than one packet).

3. *Use Quick Links*: In a federated environment, users should be instructed to ensure the server connection to the network is as fast as possible. This is due to the marked decrease in fetch time relative to the link speed.

4. *Optimize Site Submission Order*: Although no control can be exerted over remote DBMS, link speed or server CPU speed, optimizing the order that queries are submitted to remote sources may provide additional time savings. For example, a query could be submitted to a slow data source first so that the additional network time required to fetch data from that source over the time required for a faster source is minimized.

7 Conclusion

In this paper, an overview of federated database technology has been provided. Several factors motivated the development of these systems: including data sharing within organizations, company mergers and acquisitions, and geographic separation of teams. Two different models of federation were presented. The global schema model provides a unified global view of remote data sources. The multidatabase language model provides no global schema, and requires users to explicitly name the remote data sources their query will use. An overview of the Omni module within ASA was provided.

Some optimization strategies were presented, including the semijoin, replicate and Garlic optimization algorithms. Strenghts and weaknesses of each strategy were presented, and the

reader was directed to additional sources for an overview of other optimization algorithms. Some of the challenges of performing global optimization were also addressed.

There are many inputs to a global cost model. Some of the most pertinent were noted, including communication, data reformatting, and subquery and method execution costs. Some basic formulas for evaluation of these costs were derived; including a working cost model used to separate communication cost from the additional costs of performing a query.

Several applications were developed to setup and perform the outlined experiments, and to analyze the data from those experiments. An overview as well as usage instructions for DBCreate, NetBench, DBBench and ResultParse were provided.

Two sets of experiments were performed to obtain the discussed results. An overview of the hardware and software setup used to perform the experiments was presented, as well as the experimental procedure used to collect the analyzed data. A detailed analysis of the data collected was given. Some recommendations for improving Omni were discussed.

8 Future Work

There are three major areas for future research.

1. *Collection of additional data*: To enable the development of a global cost model, additional data must be collected. The experiments outlined in section 5 must be expanded to include the variance of network utilization. This will help determine what impact network traffic has on communication cost. Further tests may include increasing the computation load on each server to determine its affect on communication cost.

2. *Generation and testing of a communication cost model*: Once additional data has been gathered, it should be possible to refine all the results into a model for calculating the approximate cost of transferring data through a network. This would be useful in determining, for example, join strategies to be used when some data resides in a remote data source, and some data resides locally. Such a model would require testing on machines with different hardware configurations to determine its accuracy.

3. *Gathering and analysis of further cost model parameters*: as was outlined in section 3, there are many factors that can affect a cost model for global query optimization. Which factors can be modeled? Which factors can be instrumented? Which factors can be controlled in a global setting? Further thought on these topics is required to answer the above questions and possibly generate a more complete cost model for global query optimization.

References

[1] Sibel Adali, K. Selcuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *SIGMOD Conference*, pages 137–148, 1996.

[2] P. Bodorik, J. Pyra, and J. S. Riordon. Correcting execution of distributed queries. In *Proceedings of the second international symposium on Databases in parallel and distributed systems*, pages 192–201. ACM Press, 1990.

[3] David Brill, Marjorie Templeton, and Clement T. Yu. Distributed query processing strategies in mermaid, a frontend to data management systems. In *Proceedings of the First International Conference on Data Engineering*, pages 211–218. IEEE Computer Society, 1984.

[4] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, 1990.

[5] Neil Coburn and Per-Ake Larson. Multidatabase services: issues and architectural design. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 57–66, Toronto, Ontario, Canada, 1992. IBM Press.

[6] John Grant, Witold Litwin, Nick Roussopoulos, and Timos Sellis. Query languages for relational multidatabases. *The VLDB Journal*, 2(2):153–172, 1993.

[7] L. M. Haas, P. Kodali, J. E. Rice, P. M. Schwarz, and W. C. Swope. Integrating life sciences data-with a little garlic. In *Proceedings of the 1st IEEE International Symposium on Bioinformatics and Biomedical Engineering*, page 5. IEEE Computer Society, 2000.

[8] L. M. Haas, E. T. Lin, and M. A. Roth. Data integration through database federation. *IBM Systems Journal*, 41(4):578–596, 2002.

[9] L. M. Haas, P. M. Schwarz, P. Kodali, E. Kotlar, J. E. Rice, and W. C. Swope. Discoverylink: a system for integrated access to life sciences data sources. *IBM Syst. J.*, 40(2):489–511, 2001.

[10] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 276–285. Morgan Kaufmann Publishers Inc., 1997.

[11] David K. Hsiao. Federated databases and systems: part i — a tutorial on their data sharing. *The VLDB Journal*, 1(1):127–180, 1992.

[12] David K. Hsiao. Federated databases and systems: part ii — a tutorial on their resource consolidation. *The VLDB Journal*, 1(2):285–310, 1992.

[13] Ryan Huebsch and Shawn R. Jeffery. Freddies: Dht-based adapative query processing. Technical report, UC Berkeley, 2003.

[14] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 18–27. ACM Press, 1988.

[15] H. Lu, B. C. Ooi, and C. H. Goh. Multidatabase query optimization: Issues and solutions. In *Proceedings of Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 137–143, 1993.

[16] Hongjun Lu, Beng-Chin Ooi, and Cheng-Hian Goh. On global multidatabase query optimization. *SIGMOD Rec.*, 21(4):6–11, 1992.

[17] Sun Microsystems. J2se 1.4.2. http://java.sun.com/j2se/1.4.2/index.jsp, Accessed December 3rd, 2004.

[18] Sun Microsystems. J2se 5.0. http://java.sun.com/j2se/1.5.0/index.jsp, Accessed December 3rd, 2004.

[19] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1999.

[20] S. B. Navathe R. Elmasri. *Fundamentals of Database Systems*. Addison Wesley, 3rd edition, 2000.

[21] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *The VLDB Journal*, pages 599–610, 1999.

[22] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International*

Conference on Management of Data, Boston, Massachusetts, May 30 - June 1, pages 23–34. ACM, 1979.

[23] Leonard D. Shapiro. Join processing in database systems with large main memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[24] Shashi Shekhar, Jaideep Srivastava, and Soumitra Dutta. A formal model of trade-off between optimization and execution costs in semantic query optimization. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 457–467. Morgan Kaufmann Publishers Inc., 1988.

[25] Inc. Sybase. *SQL Anywhere Studio 9.0.2*, 2004.

[26] Marjorie Templeton, Herbert Henley, Edward Maros, and Darrel J. Van Buer. Interviso: dealing with the complexity of federated database access. *The VLDB Journal*, 4(2):287–318, 1995.

[27] H. J. A. van Kuijk, F. H. E. Pijpers, and Peter M. G. Apers. Semantic query optimization in distributed databases. In *International Conference Proceedings of Advances in Computing and Information*, pages 295–303, May 1990.

[28] Weipeng Paul Yan. Interchanging group-by and join in distributed query processing. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 823–831. IBM Press, 1993.

[29] C. T. Yu, L. Lilien, K. C. Guh, and M. Templeton. Adaptive techniques for distributed query optimization. In *IEEE 1986 International Conference on Data Engineering*, pages 86–93, 1986.

[30] Qiang Zhu. Query optimization in multidatabase systems. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 111–127. IBM Press, 1992.

[31] Qiang Zhu. *Estimating Local Cost Parameters for Global Query Optimization in a Multidatabase System*. PhD thesis, University of Waterloo, 1995.

[32] Qiang Zhu and Per-Ake Larson. Solving local cost estimation problem for global query optimization in multidatabase systems. *Distrib. Parallel Databases*, 6(4):373–421, 1998.

A NetBench Configurations

NetBench configurations are shown in table 15. These configurations were used when running experiment 1 in section 5.

Configuration	MODE	BYTES	PACKETS	RUNS
1	1	2	50 000	30
2	1	3	50 000	30
3	1	4	50 000	30
4	1	5	50 000	30
5	1	6	50 000	30
6	1	7	50 000	30
7	1	8	50 000	30
8	1	9	50 000	30
9	1	10	50 000	30
10	1	50	50 000	30
11	1	75	50 000	30
12	1	100	50 000	30
13	1	125	50 000	30
14	1	150	50 000	30
15	1	175	50 000	30
16	1	225	50 000	30
17	1	250	50 000	30
18	1	200	50 000	30
19	1	300	50 000	30
20	1	400	50 000	30
21	1	500	50 000	30
22	1	600	50 000	30
23	1	700	50 000	30
24	1	800	50 000	30
25	1	900	50 000	30
26	1	1000	50 000	30

Table 15: NetBench Configurations

B Queries

This appendix contains those queries mentioned in section 5.

```
SELECT * FROM table_char2;
SELECT max(col1) FROM table_char2;
SELECT max(col25) FROM table_char2;
SELECT max(col100) FROM table_char2;
SELECT col1 FROM table_char2;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_char2;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_char2;
```

Figure 11: Query 2

```
SELECT * FROM table_char3;
SELECT max(col1) FROM table_char3;
SELECT max(col25) FROM table_char3;
SELECT max(col100) FROM table_char3;
SELECT col1 FROM table_char3;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_char3;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_char3;
```

Figure 12: Query 3

```
SELECT * FROM table_char4;
SELECT max(col1) FROM table_char4;
SELECT max(col25) FROM table_char4;
SELECT max(col100) FROM table_char4;
SELECT col1 FROM table_char4;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_char4;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_char4;
```

Figure 13: Query 4

```
SELECT * FROM table_char5;
SELECT max(col1) FROM table_char5;
SELECT max(col25) FROM table_char5;
SELECT max(col100) FROM table_char5;
SELECT col1 FROM table_char5;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_char5;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_char5;
```

Figure 14: Query 5

```
SELECT * FROM table_char6;
SELECT max(col1) FROM table_char6;
SELECT max(col25) FROM table_char6;
SELECT max(col100) FROM table_char6;
SELECT col1 FROM table_char6;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_char6;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_char6;
```

Figure 15: Query 6

```
SELECT * FROM table_char7;
SELECT max(col1) FROM table_char7;
SELECT max(col25) FROM table_char7;
SELECT max(col100) FROM table_char7;
SELECT col1 FROM table_char7;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_char7;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_char7;
```

Figure 16: Query 7

```
SELECT * FROM table_char8;
SELECT max(col1) FROM table_char8;
SELECT max(col25) FROM table_char8;
SELECT max(col100) FROM table_char8;
SELECT col1 FROM table_char8;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_char8;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_char8;
```

Figure 17: Query 8

```
SELECT * FROM table_char9;
SELECT max(col1) FROM table_char9;
SELECT max(col25) FROM table_char9;
SELECT max(col100) FROM table_char9;
SELECT col1 FROM table_char9;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_char9;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_char9;
```

Figure 18: Query 9

```
SELECT * FROM table_char10;
SELECT max(col1) FROM table_char10;
SELECT max(col25) FROM table_char10;
SELECT max(col100) FROM table_char10;
SELECT col1 FROM table_char10;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_char10;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_char10;
```

Figure 19: Query 10

```
SELECT * FROM table_varchar2;
SELECT max(col1) FROM table_varchar2;
SELECT max(col25) FROM table_varchar2;
SELECT max(col100) FROM table_varchar2;
SELECT col1 FROM table_varchar2;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_varchar2;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_varchar2;
```

Figure 20: Query 12

```
SELECT * FROM table_varchar3;
SELECT max(col1) FROM table_varchar3;
SELECT max(col25) FROM table_varchar3;
SELECT max(col100) FROM table_varchar3;
SELECT col1 FROM table_varchar3;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_varchar3;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_varchar3;
```

Figure 21: Query 13

```
SELECT * FROM table_varchar4;
SELECT max(col1) FROM table_varchar4;
SELECT max(col25) FROM table_varchar4;
SELECT max(col100) FROM table_varchar4;
SELECT col1 FROM table_varchar4;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_varchar4;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_varchar4;
```

Figure 22: Query 14

```
SELECT * FROM table_varchar5;
SELECT max(col1) FROM table_varchar5;
SELECT max(col25) FROM table_varchar5;
SELECT max(col100) FROM table_varchar5;
SELECT col1 FROM table_varchar5;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_varchar5;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_varchar5;
```

Figure 23: Query 15

```
SELECT * FROM table_varchar6;
SELECT max(col1) FROM table_varchar6;
SELECT max(col25) FROM table_varchar6;
SELECT max(col100) FROM table_varchar6;
SELECT col1 FROM table_varchar6;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_varchar6;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_varchar6;
```

Figure 24: Query 16

```
SELECT * FROM table_varchar7;
SELECT max(col1) FROM table_varchar7;
SELECT max(col25) FROM table_varchar7;
SELECT max(col100) FROM table_varchar7;
SELECT col1 FROM table_varchar7;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_varchar7;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_varchar7;
```

Figure 25: Query 17

```
SELECT * FROM table_varchar8;
SELECT max(col1) FROM table_varchar8;
SELECT max(col25) FROM table_varchar8;
SELECT max(col100) FROM table_varchar8;
SELECT col1 FROM table_varchar8;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_varchar8;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_varchar8;
```

Figure 26: Query 18

```
SELECT * FROM table_varchar9;
SELECT max(col1) FROM table_varchar9;
SELECT max(col25) FROM table_varchar9;
SELECT max(col100) FROM table_varchar9;
SELECT col1 FROM table_varchar9;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_varchar9;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_varchar9;
```

Figure 27: Query 19

```
SELECT * FROM table_varchar10;
SELECT max(col1) FROM table_varchar10;
SELECT max(col25) FROM table_varchar10;
SELECT max(col100) FROM table_varchar10;
SELECT col1 FROM table_varchar10;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_varchar10;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_varchar10;
```

Figure 28: Query 20

```
SELECT * FROM table_date;
SELECT max(col1) FROM table_date;
SELECT max(col25) FROM table_date;
SELECT max(col100) FROM table_date;
SELECT col1 FROM table_date;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_date;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28,
col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42,
col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56,
col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70,
col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84,
col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98,
col99, col100 FROM table_date;
```

Figure 29: Query 30

```
SELECT * FROM table_decimal;
SELECT max(col1) FROM table_decimal;
SELECT max(col25) FROM table_decimal;
SELECT max(col100) FROM table_decimal;
SELECT col1 FROM table_decimal;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_decimal;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_decimal;
```

Figure 30: Query 31

```
SELECT * FROM table_double;
SELECT max(col1) FROM table_double;
SELECT max(col25) FROM table_double;
SELECT max(col100) FROM table_double;
SELECT col1 FROM table_double;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_double;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28, col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42, col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56, col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70, col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84, col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98, col99, col100 FROM table_double;
```

Figure 31: Query 32

```
SELECT * FROM table_float;
SELECT max(col1) FROM table_float;
SELECT max(col25) FROM table_float;
SELECT max(col100) FROM table_float;
SELECT col1 FROM table_float;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_float;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28,
col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42,
col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56,
col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70,
col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84,
col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98,
col99, col100 FROM table_float;
```

Figure 32: Query 33

```
SELECT * FROM table_int;
SELECT max(col1) FROM table_int;
SELECT max(col25) FROM table_int;
SELECT max(col100) FROM table_int;
SELECT col1 FROM table_int;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_int;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28,
col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42,
col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56,
col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70,
col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84,
col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98,
col99, col100 FROM table_int;
```

Figure 33: Query 34

```
SELECT * FROM table_real;
SELECT max(col1) FROM table_real;
SELECT max(col25) FROM table_real;
SELECT max(col100) FROM table_real;
SELECT col1 FROM table_real;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_real;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28,
col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42,
col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56,
col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70,
col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84,
col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98,
col99, col100 FROM table_real;
```

Figure 34: Query 35

```
SELECT * FROM table_smallint;
SELECT max(col1) FROM table_smallint;
SELECT max(col25) FROM table_smallint;
SELECT max(col100) FROM table_smallint;
SELECT col1 FROM table_smallint;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13,
col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM
table_smallint;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28,
col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42,
col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56,
col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70,
col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84,
col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98,
col99, col100 FROM table_smallint;
```

Figure 35: Query 36

```
SELECT * FROM table_time;
SELECT max(col1) FROM table_time;
SELECT max(col25) FROM table_time;
SELECT max(col100) FROM table_time;
SELECT col1 FROM table_time;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM table_time;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28,
col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42,
col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56,
col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70,
col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84,
col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98,
col99, col100 FROM table_time;
```

Figure 36: Query 37

```
SELECT * FROM table_timestamp;
SELECT max(col1) FROM table_timestamp;
SELECT max(col25) FROM table_timestamp;
SELECT max(col100) FROM table_timestamp;
SELECT col1 FROM table_timestamp;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13,
col14, col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25 FROM
table_timestamp;
SELECT col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14,
col15, col16, col17, col18, col19, col20, col21, col22, col23, col24, col25, col26, col27, col28,
col29, col30, col31, col32, col33, col34, col35, col36, col37, col38, col39, col40, col41, col42,
col43, col44, col45, col46, col47, col48, col49, col50, col51, col52, col53, col54, col55, col56,
col57, col58, col59, col60, col61, col62, col63, col64, col65, col66, col67, col68, col69, col70,
col71, col72, col73, col74, col75, col76, col77, col78, col79, col80, col81, col82, col83, col84,
col85, col86, col87, col88, col89, col90, col91, col92, col93, col94, col95, col96, col97, col98,
col99, col100 FROM table_timestamp;
```

Figure 37: Query 38

C Schemas

This appendix contains those schemas required by the queries performed in section 5. The schemas are transformed in order to conform to the SQL dialect of the system they are submitted to (i.e. schemas are translated into the SQL dialect before being created in each of the systems).

```
CREATE   TABLE    table_char2 (
         col1     CHAR(2),
         col2     CHAR(2),
         col3     CHAR(2),
         col4     CHAR(2),
         col5     CHAR(2),
         col6     CHAR(2),
         col7     CHAR(2),
         col8     CHAR(2),
         ⋮        ⋮
         col99    CHAR(2),
         col100   CHAR(2)
)
```

Figure 38: Schema for Table type_char2

```
CREATE   TABLE    table_char3 (
         col1     CHAR(3),
         col2     CHAR(3),
         col3     CHAR(3),
         col4     CHAR(3),
         col5     CHAR(3),
         col6     CHAR(3),
         col7     CHAR(3),
         col8     CHAR(3),
         ⋮        ⋮
         col99    CHAR(3),
         col100   CHAR(3)
)
```

Figure 39: Schema for Table type_char3

```
CREATE   TABLE    table_char4 (
         col1     CHAR(4),
         col2     CHAR(4),
         col3     CHAR(4),
         col4     CHAR(4),
         col5     CHAR(4),
         col6     CHAR(4),
         col7     CHAR(4),
         col8     CHAR(4),
         ⋮        ⋮
         col99    CHAR(4),
         col100   CHAR(4)
)
```

Figure 40: Schema for Table type_char4

```
CREATE   TABLE    table_char5 (
         col1     CHAR(5),
         col2     CHAR(5),
         col3     CHAR(5),
         col4     CHAR(5),
         col5     CHAR(5),
         col6     CHAR(5),
         col7     CHAR(5),
         col8     CHAR(5),
         ⋮        ⋮
         col99    CHAR(5),
         col100   CHAR(5)
)
```

Figure 41: Schema for Table type_char5

```
CREATE   TABLE    table_char6 (
         col1     CHAR(6),
         col2     CHAR(6),
         col3     CHAR(6),
         col4     CHAR(6),
         col5     CHAR(6),
         col6     CHAR(6),
         col7     CHAR(6),
         col8     CHAR(6),
         ⋮        ⋮
         col99    CHAR(6),
         col100   CHAR(6)
)
```

Figure 42: Schema for Table type_char6

```
CREATE   TABLE    table_char7 (
         col1     CHAR(7),
         col2     CHAR(7),
         col3     CHAR(7),
         col4     CHAR(7),
         col5     CHAR(7),
         col6     CHAR(7),
         col7     CHAR(7),
         col8     CHAR(7),
         ⋮        ⋮
         col99    CHAR(7),
         col100   CHAR(7)
)
```

Figure 43: Schema for Table type_char7

```
CREATE   TABLE    table_char8 (
         col1     CHAR(8),
         col2     CHAR(8),
         col3     CHAR(8),
         col4     CHAR(8),
         col5     CHAR(8),
         col6     CHAR(8),
         col7     CHAR(8),
         col8     CHAR(8),
         ⋮        ⋮
         col99    CHAR(8),
         col100   CHAR(8)
)
```

Figure 44: Schema for Table type_char8

```
CREATE   TABLE    table_char9 (
         col1     CHAR(9),
         col2     CHAR(9),
         col3     CHAR(9),
         col4     CHAR(9),
         col5     CHAR(9),
         col6     CHAR(9),
         col7     CHAR(9),
         col8     CHAR(9),
         ⋮        ⋮
         col99    CHAR(9),
         col100   CHAR(9)
)
```

Figure 45: Schema for Table type_char9

```
CREATE   TABLE    table_char10 (
         col1     CHAR(10),
         col2     CHAR(10),
         col3     CHAR(10),
         col4     CHAR(10),
         col5     CHAR(10),
         col6     CHAR(10),
         col7     CHAR(10),
         col8     CHAR(10),
         ⋮        ⋮
         col99    CHAR(10),
         col100   CHAR(10)
)
```

Figure 46: Schema for Table type_char10

```
CREATE   TABLE    table_date (
         col1     DATE,
         col2     DATE,
         col3     DATE,
         col4     DATE,
         col5     DATE,
         col6     DATE,
         col7     DATE,
         col8     DATE,
         ⋮        ⋮
         col99    DATE,
         col100   DATE
)
```

Figure 47: Schema for Table type_date

```
CREATE   TABLE    table_decimal (
         col1     DECIMAL,
         col2     DECIMAL,
         col3     DECIMAL,
         col4     DECIMAL,
         col5     DECIMAL,
         col6     DECIMAL,
         col7     DECIMAL,
         col8     DECIMAL,
         ⋮        ⋮
         col99    DECIMAL,
         col100   DECIMAL
)
```

Figure 48: Schema for Table type_decimal

```
CREATE   TABLE    table_double (
         col1     DOUBLE,
         col2     DOUBLE,
         col3     DOUBLE,
         col4     DOUBLE,
         col5     DOUBLE,
         col6     DOUBLE,
         col7     DOUBLE,
         col8     DOUBLE,
         ⋮        ⋮
         col99    DOUBLE,
         col100   DOUBLE
)
```

Figure 49: Schema for Table type_double

```
CREATE   TABLE    table_float (
         col1     FLOAT,
         col2     FLOAT,
         col3     FLOAT,
         col4     FLOAT,
         col5     FLOAT,
         col6     FLOAT,
         col7     FLOAT,
         col8     FLOAT,
         ⋮        ⋮
         col99    FLOAT,
         col100   FLOAT
)
```

Figure 50: Schema for Table type_float

```
CREATE  TABLE   table_int (
        col1    INT,
        col2    INT,
        col3    INT,
        col4    INT,
        col5    INT,
        col6    INT,
        col7    INT,
        col8    INT,
        ⋮       ⋮
        col99   INT,
        col100  INT
)
```

Figure 51: Schema for Table type_int

```
CREATE  TABLE   table_real (
        col1    REAL,
        col2    REAL,
        col3    REAL,
        col4    REAL,
        col5    REAL,
        col6    REAL,
        col7    REAL,
        col8    REAL,
        ⋮       ⋮
        col99   REAL,
        col100  REAL
)
```

Figure 52: Schema for Table type_real

```
CREATE   TABLE    table_smallint (
         col1     SMALLINT,
         col2     SMALLINT,
         col3     SMALLINT,
         col4     SMALLINT,
         col5     SMALLINT,
         col6     SMALLINT,
         col7     SMALLINT,
         col8     SMALLINT,
         ⋮        ⋮
         col99    SMALLINT,
         col100   SMALLINT
)
```

Figure 53: Schema for Table type_smallint

```
CREATE   TABLE    table_time (
         col1     TIME,
         col2     TIME,
         col3     TIME,
         col4     TIME,
         col5     TIME,
         col6     TIME,
         col7     TIME,
         col8     TIME,
         ⋮        ⋮
         col99    TIME,
         col100   TIME
)
```

Figure 54: Schema for Table type_time

```
CREATE   TABLE    table_timestamp (
         col1     TIMESTAMP,
         col2     TIMESTAMP,
         col3     TIMESTAMP,
         col4     TIMESTAMP,
         col5     TIMESTAMP,
         col6     TIMESTAMP,
         col7     TIMESTAMP,
         col8     TIMESTAMP,
         ⋮        ⋮
         col99    TIMESTAMP,
         col100   TIMESTAMP
)
```

Figure 55: Schema for Table type_timestamp

```
CREATE   TABLE    table_varchar2 (
         col1     VARCHAR(2),
         col2     VARCHAR(2),
         col3     VARCHAR(2),
         col4     VARCHAR(2),
         col5     VARCHAR(2),
         col6     VARCHAR(2),
         col7     VARCHAR(2),
         col8     VARCHAR(2),
         ⋮        ⋮
         col99    VARCHAR(2),
         col100   VARCHAR(2)
)
```

Figure 56: Schema for Table type_varchar2

```
CREATE  TABLE   table_varchar3 (
        col1    VARCHAR(3),
        col2    VARCHAR(3),
        col3    VARCHAR(3),
        col4    VARCHAR(3),
        col5    VARCHAR(3),
        col6    VARCHAR(3),
        col7    VARCHAR(3),
        col8    VARCHAR(3),
        ⋮       ⋮
        col99   VARCHAR(3),
        col100  VARCHAR(3)
)
```

Figure 57: Schema for Table type_varchar3

```
CREATE  TABLE   table_varchar4 (
        col1    VARCHAR(4),
        col2    VARCHAR(4),
        col3    VARCHAR(4),
        col4    VARCHAR(4),
        col5    VARCHAR(4),
        col6    VARCHAR(4),
        col7    VARCHAR(4),
        col8    VARCHAR(4),
        ⋮       ⋮
        col99   VARCHAR(4),
        col100  VARCHAR(4)
)
```

Figure 58: Schema for Table type_varchar4

```
CREATE  TABLE   table_varchar5 (
        col1    VARCHAR(5),
        col2    VARCHAR(5),
        col3    VARCHAR(5),
        col4    VARCHAR(5),
        col5    VARCHAR(5),
        col6    VARCHAR(5),
        col7    VARCHAR(5),
        col8    VARCHAR(5),
        ⋮       ⋮
        col99   VARCHAR(5),
        col100  VARCHAR(5)
)
```

Figure 59: Schema for Table type_varchar5

```
CREATE  TABLE   table_varchar6 (
        col1    VARCHAR(6),
        col2    VARCHAR(6),
        col3    VARCHAR(6),
        col4    VARCHAR(6),
        col5    VARCHAR(6),
        col6    VARCHAR(6),
        col7    VARCHAR(6),
        col8    VARCHAR(6),
        ⋮       ⋮
        col99   VARCHAR(6),
        col100  VARCHAR(6)
)
```

Figure 60: Schema for Table type_varchar6

```
CREATE   TABLE    table_varchar7 (
         col1     VARCHAR(7),
         col2     VARCHAR(7),
         col3     VARCHAR(7),
         col4     VARCHAR(7),
         col5     VARCHAR(7),
         col6     VARCHAR(7),
         col7     VARCHAR(7),
         col8     VARCHAR(7),
         ⋮        ⋮
         col99    VARCHAR(7),
         col100   VARCHAR(7)
)
```

Figure 61: Schema for Table type_varchar7

```
CREATE   TABLE    table_varchar8 (
         col1     VARCHAR(8),
         col2     VARCHAR(8),
         col3     VARCHAR(8),
         col4     VARCHAR(8),
         col5     VARCHAR(8),
         col6     VARCHAR(8),
         col7     VARCHAR(8),
         col8     VARCHAR(8),
         ⋮        ⋮
         col99    VARCHAR(8),
         col100   VARCHAR(8)
)
```

Figure 62: Schema for Table type_varchar8

```
CREATE  TABLE   table_varchar9 (
        col1    VARCHAR(9),
        col2    VARCHAR(9),
        col3    VARCHAR(9),
        col4    VARCHAR(9),
        col5    VARCHAR(9),
        col6    VARCHAR(9),
        col7    VARCHAR(9),
        col8    VARCHAR(9),
        ⋮       ⋮
        col99   VARCHAR(9),
        col100  VARCHAR(9)
)
```

Figure 63: Schema for Table type_varchar9

```
CREATE  TABLE   table_varchar10 (
        col1    VARCHAR(10),
        col2    VARCHAR(10),
        col3    VARCHAR(10),
        col4    VARCHAR(10),
        col5    VARCHAR(10),
        col6    VARCHAR(10),
        col7    VARCHAR(10),
        col8    VARCHAR(10),
        ⋮       ⋮
        col99   VARCHAR(10),
        col100  VARCHAR(10)
)
```

Figure 64: Schema for Table type_varchar10

D Graphs

This section contains those figures mentioned in section 6. Both scatter plots and bar charts are provided below. The data is labelled as follows in the graph legends:

$$< ServerCPUSpeed > - < PrefetchStatus > - < LinkSpeed > \tag{13}$$

So, for example, a data label of '800-0-10' refers to a server with an 800 MHz CPU connected to a 10 Base-T link with prefetch disabled.

D.1 Scatter Plots

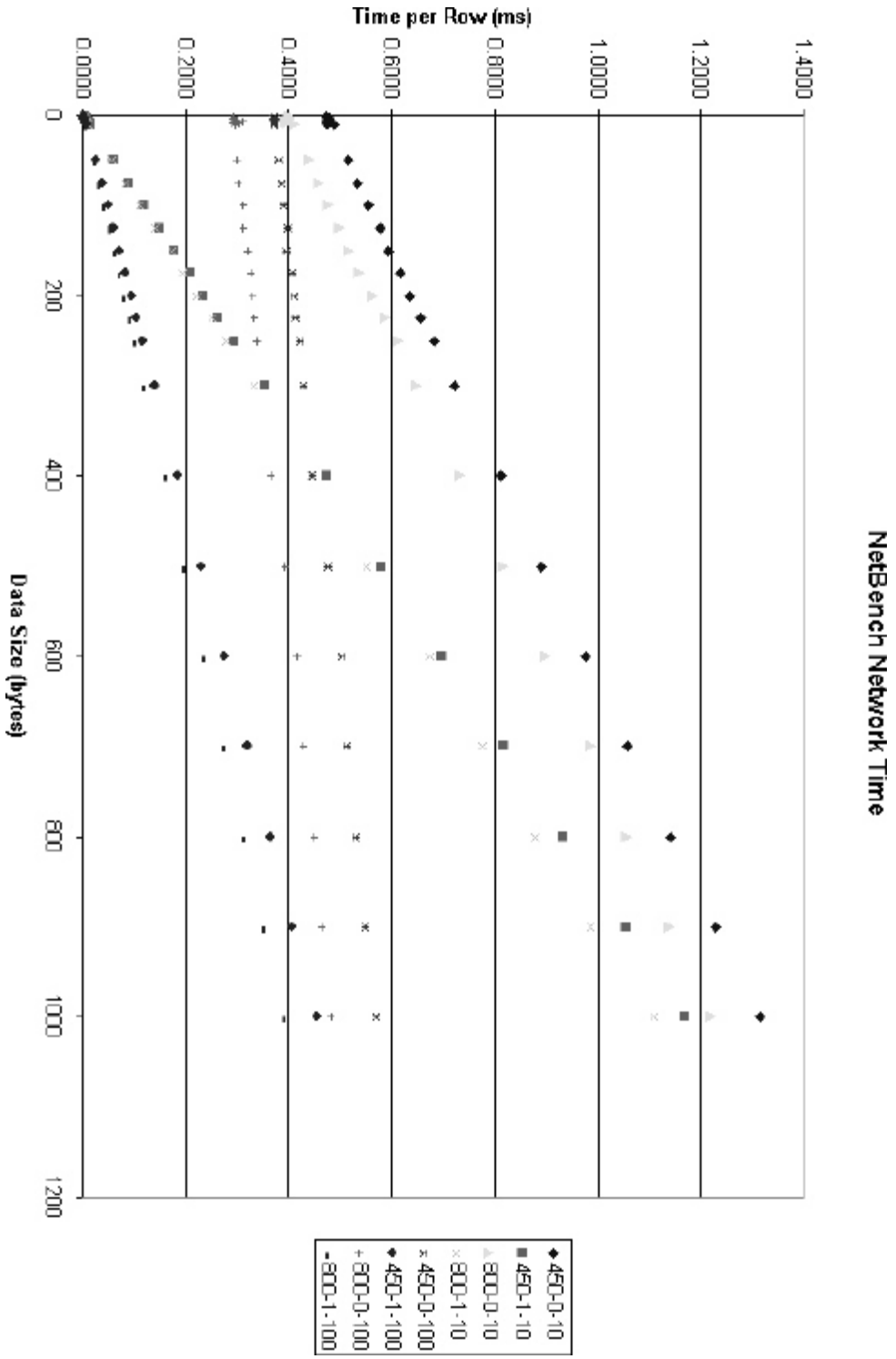

Figure 65: NetBench Network Time with 10 and 100 Base-T links and a packet size of 1460 bytes.

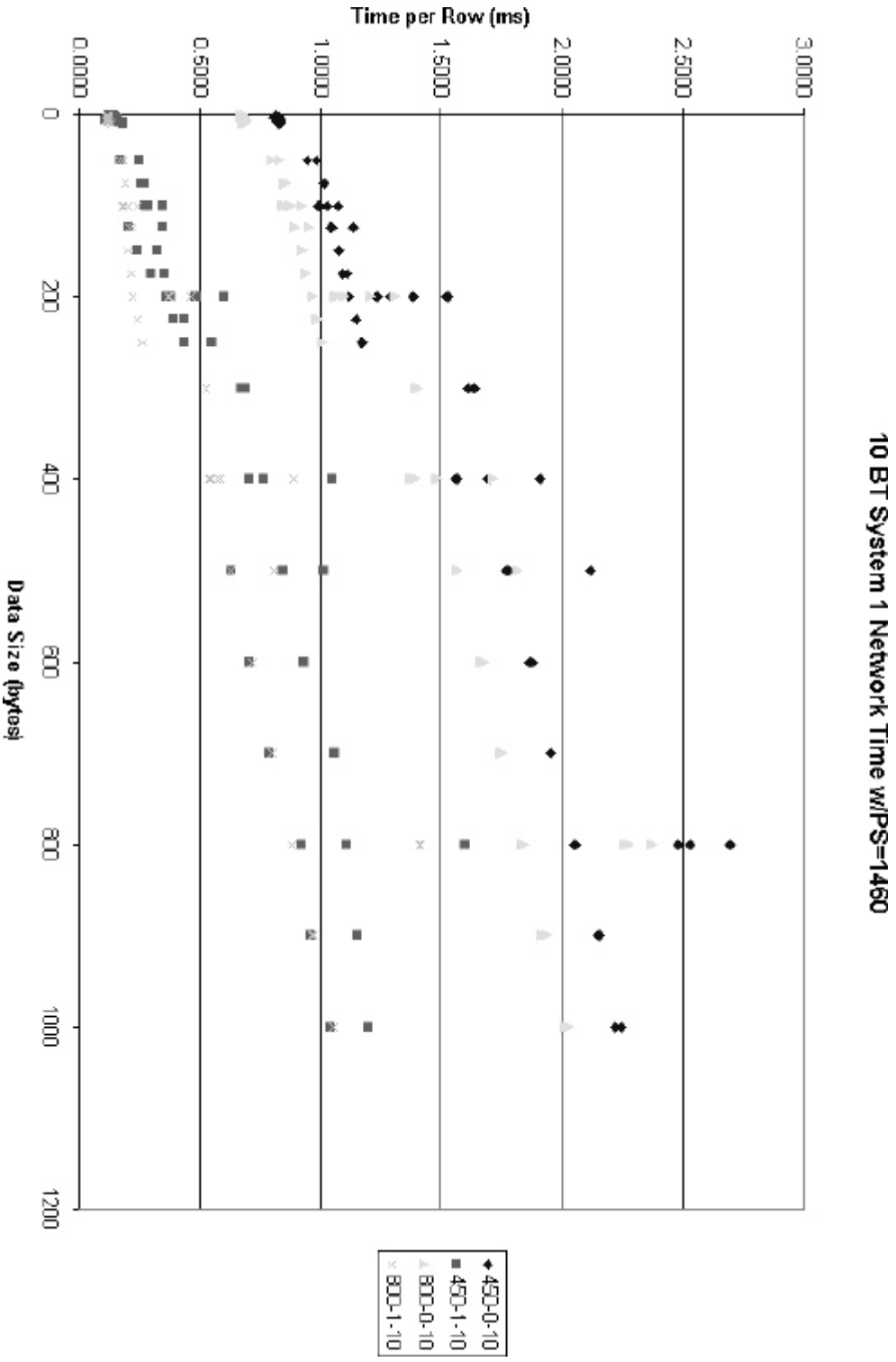

Figure 66: System 1 Network Time with a 10 Base-T link and a packet size of 1460 bytes.

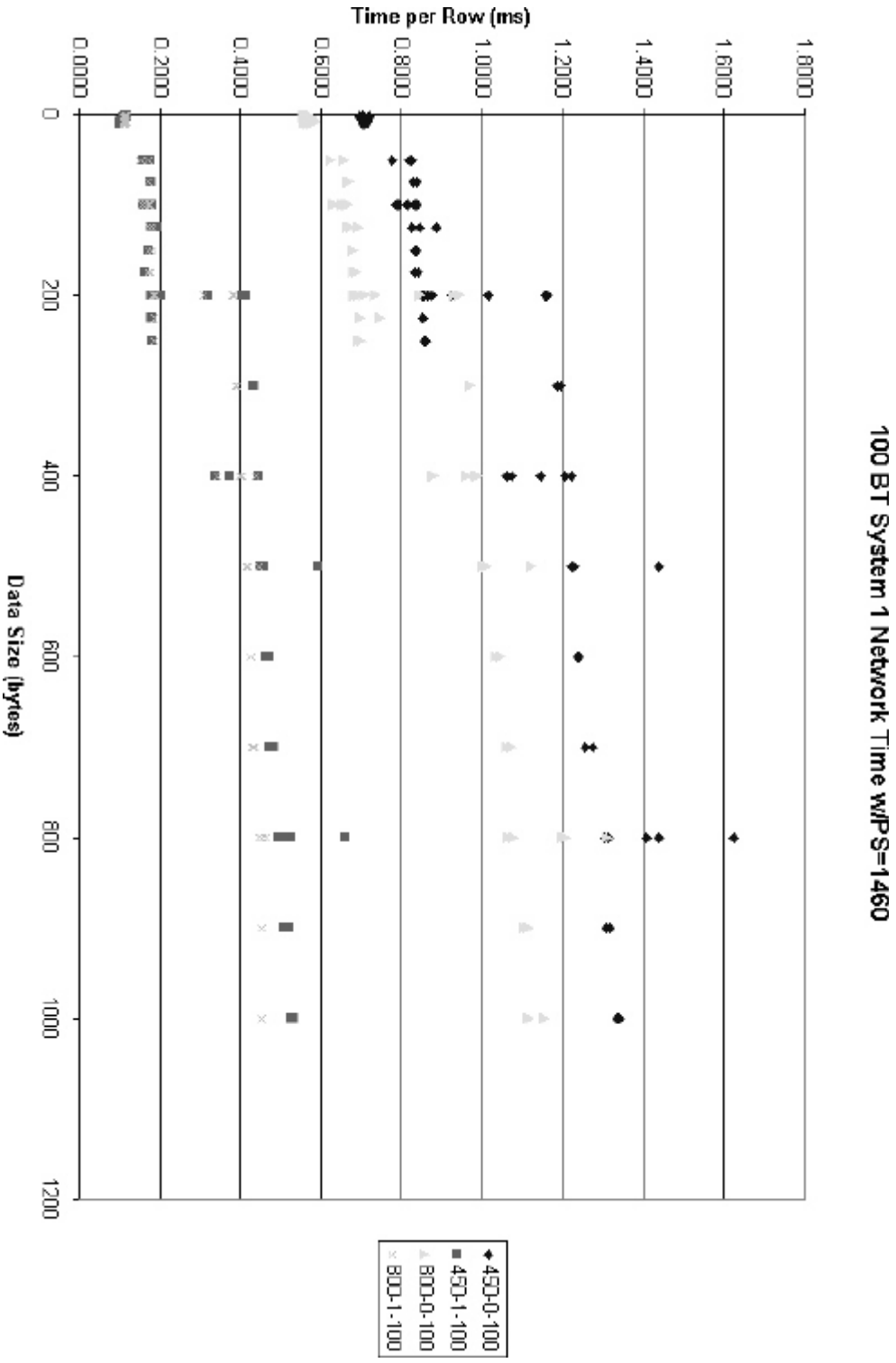

Figure 67: System 1 Network Time with a 100 Base-T link and a packet size of 1460 bytes.

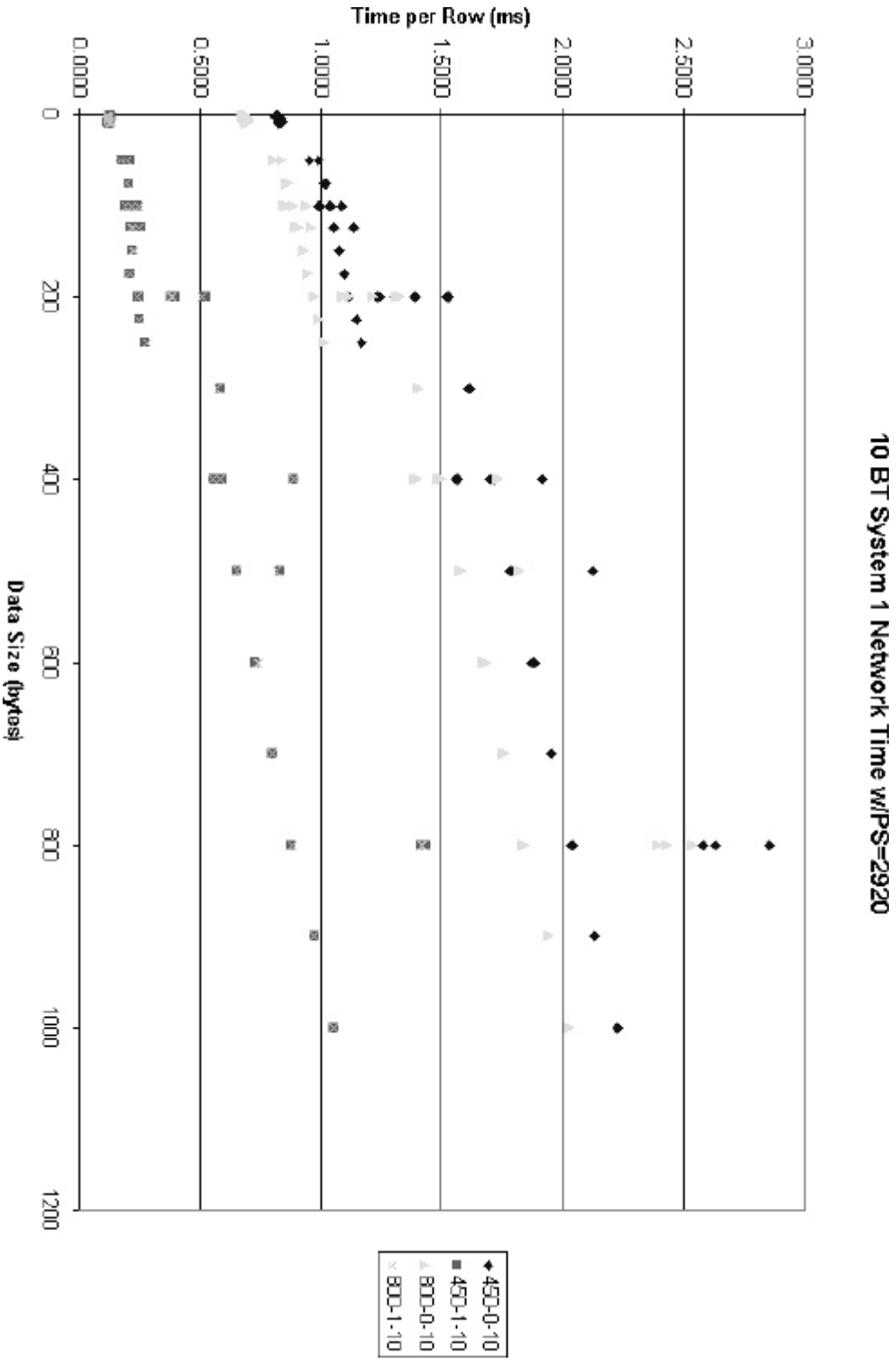

Figure 68: System 1 Network Time with a 10 Base-T link and a packet size of 2920 bytes.

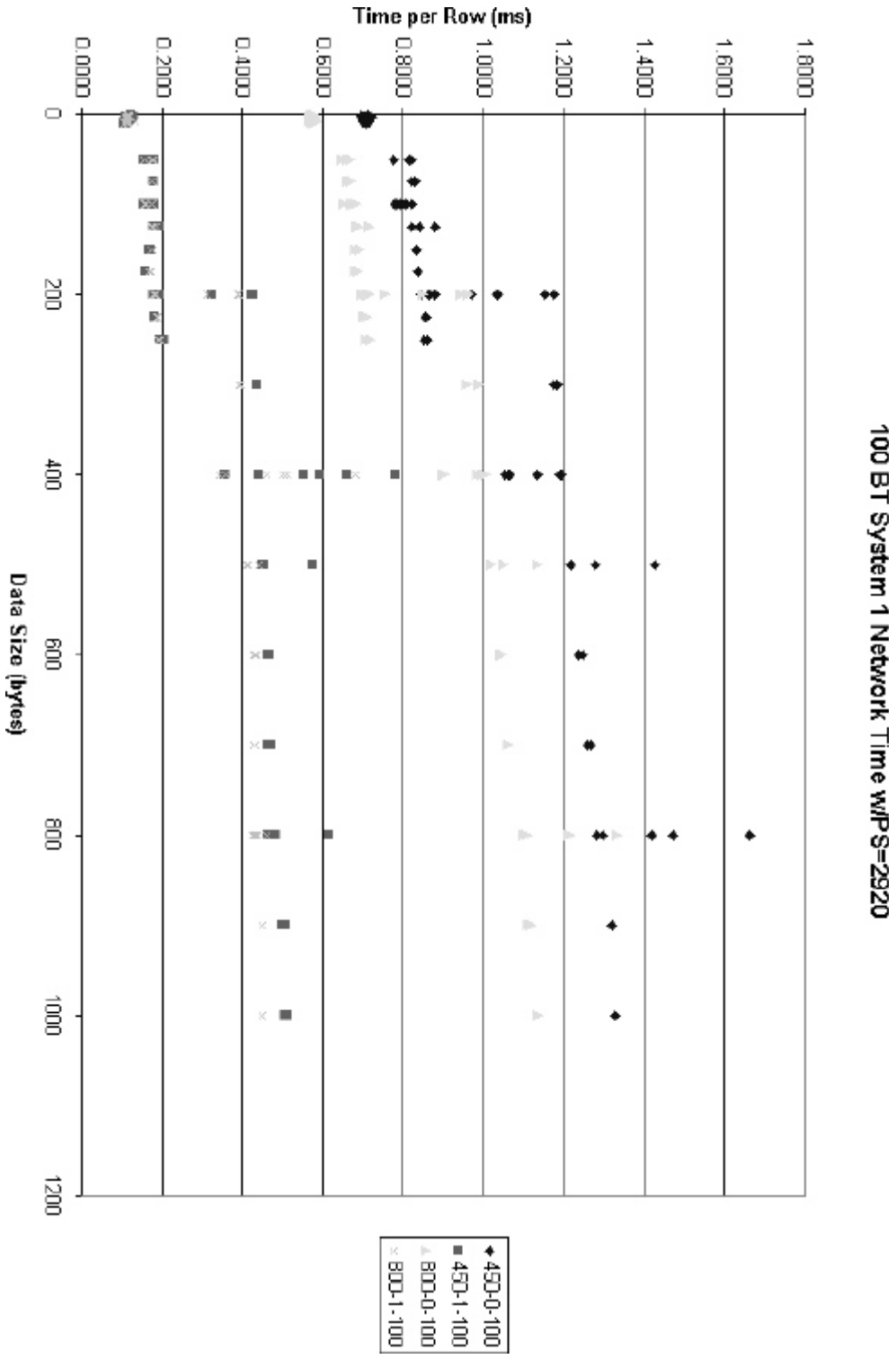

Figure 69: System 1 Network Time with a 100 Base-T link and a packet size of 2920 bytes.

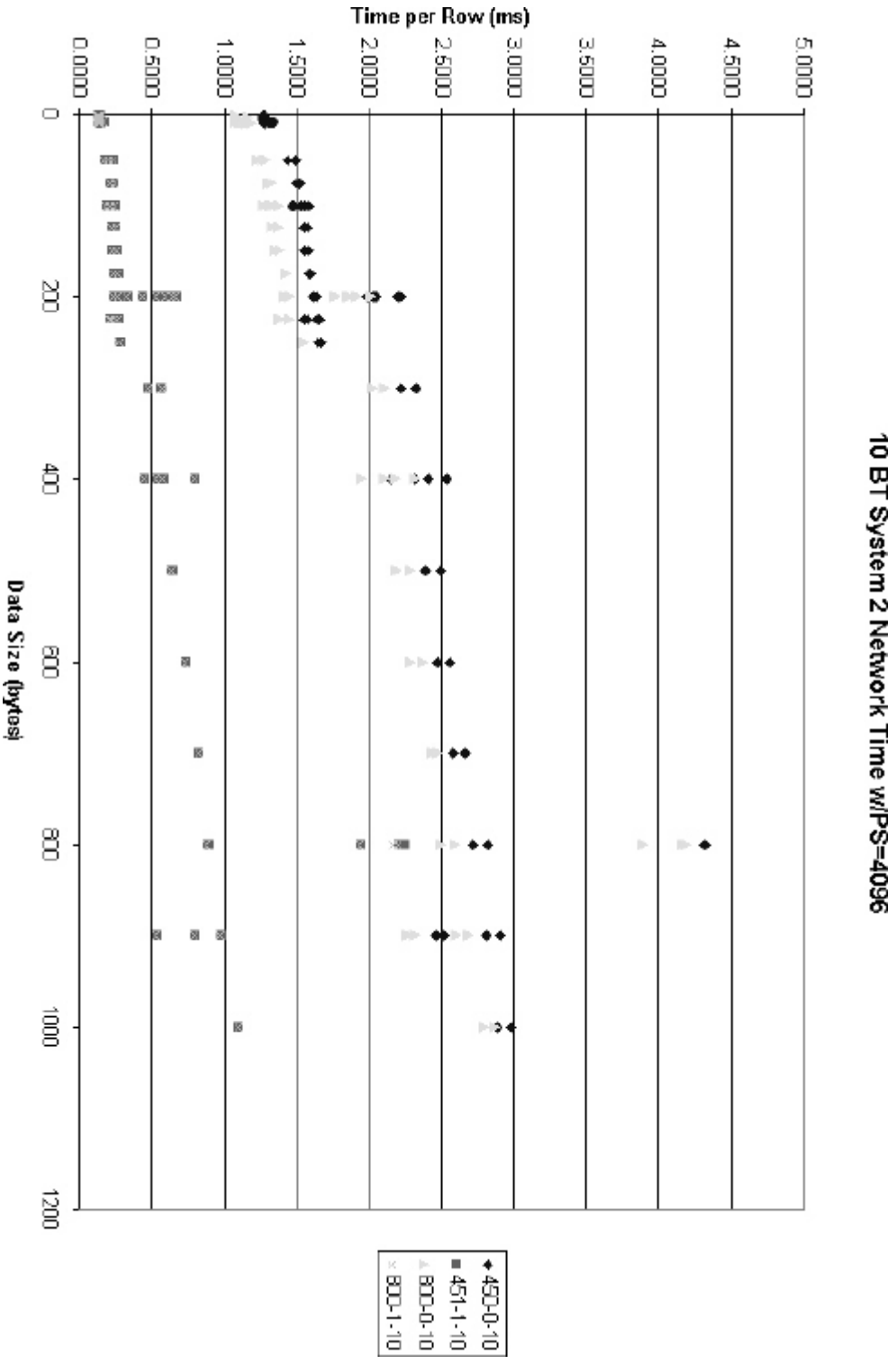

Figure 70: System 2 Network Time with a 10 Base-T link and a packet size of 4096 bytes.

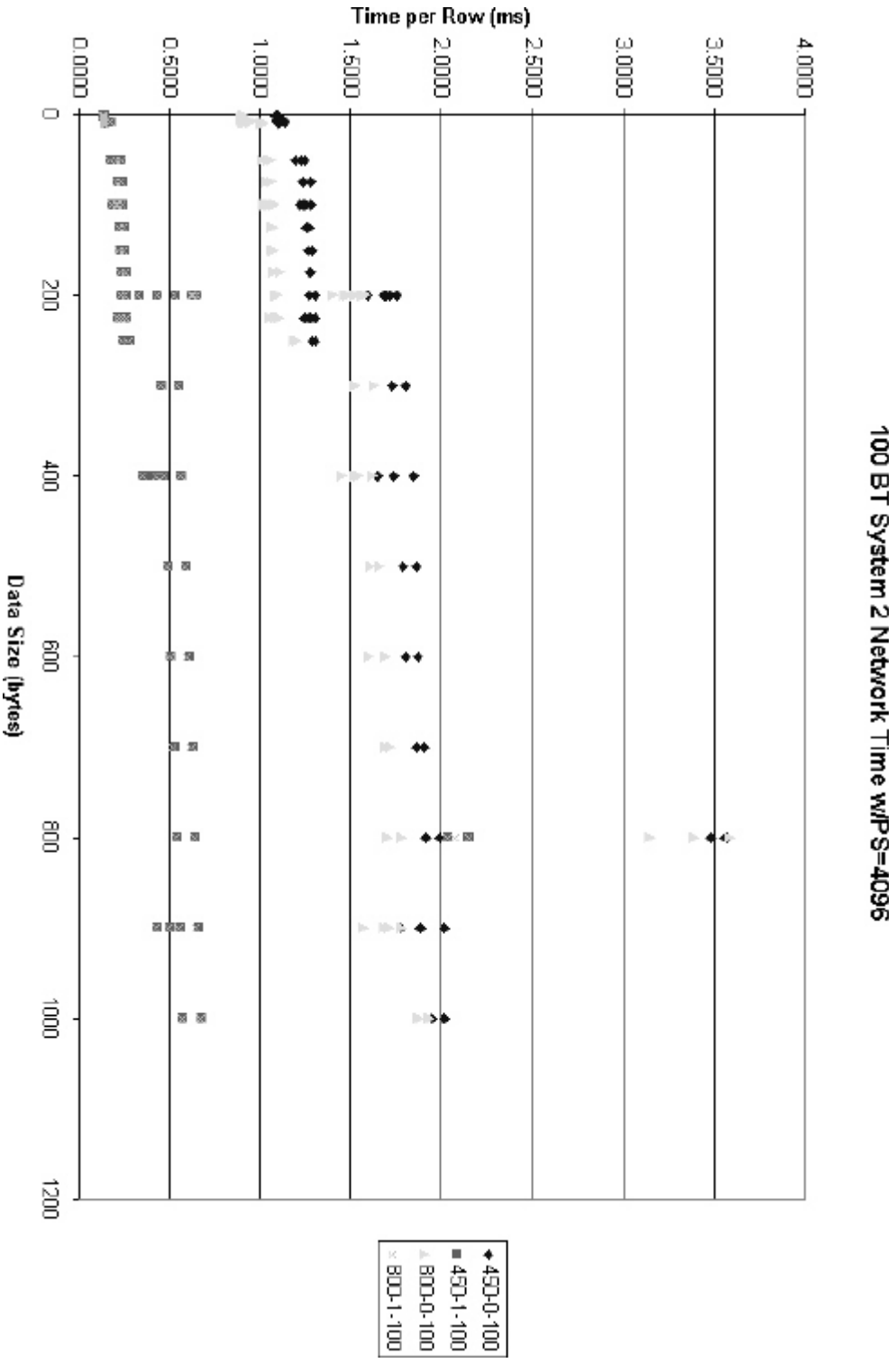

Figure 71: System 2 Network Time with a 100 Base-T link and a packet size of 4096 bytes.

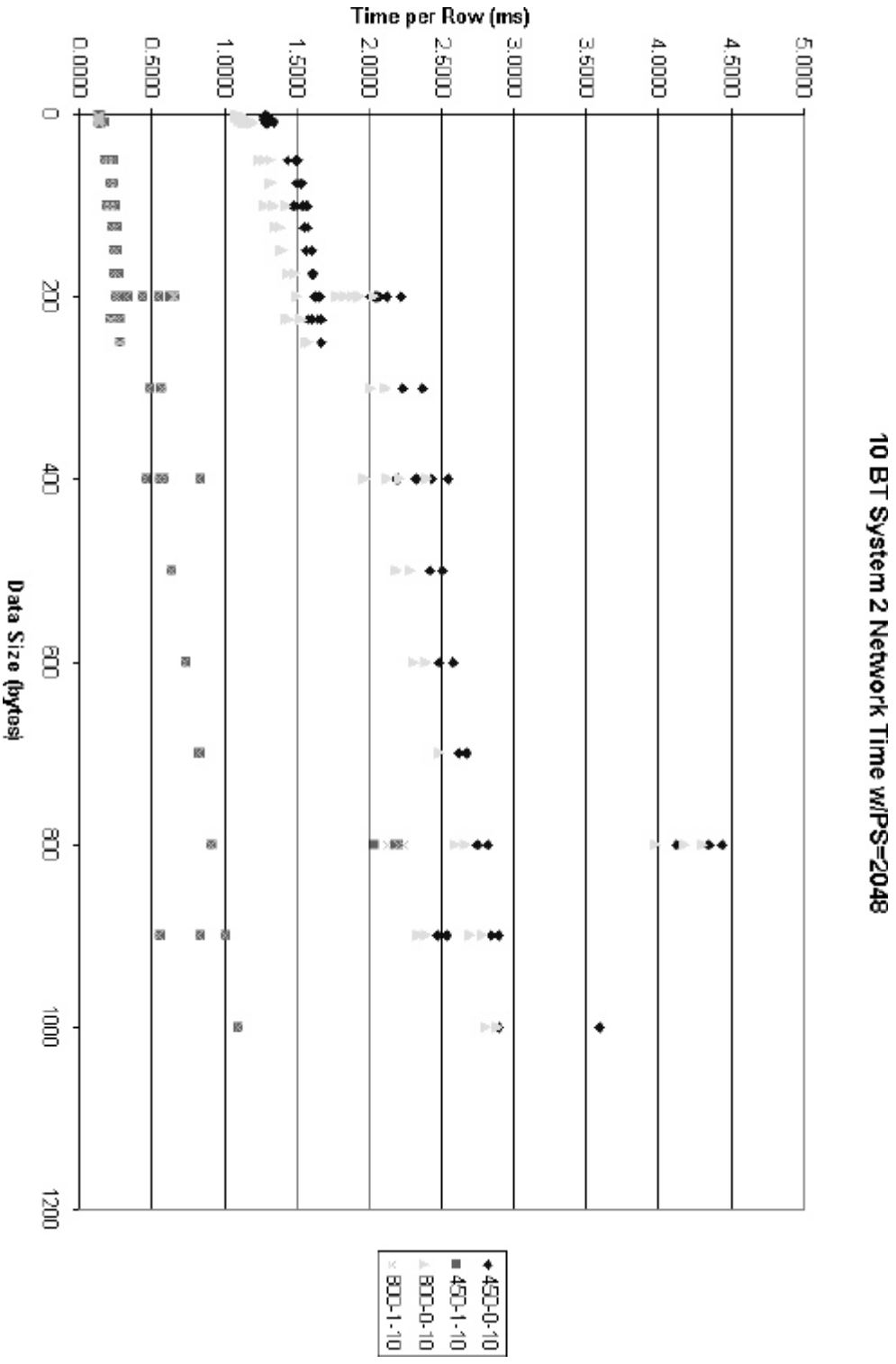

Figure 72: System 2 Network Time with a 10 Base-T link and a packet size of 2048 bytes.

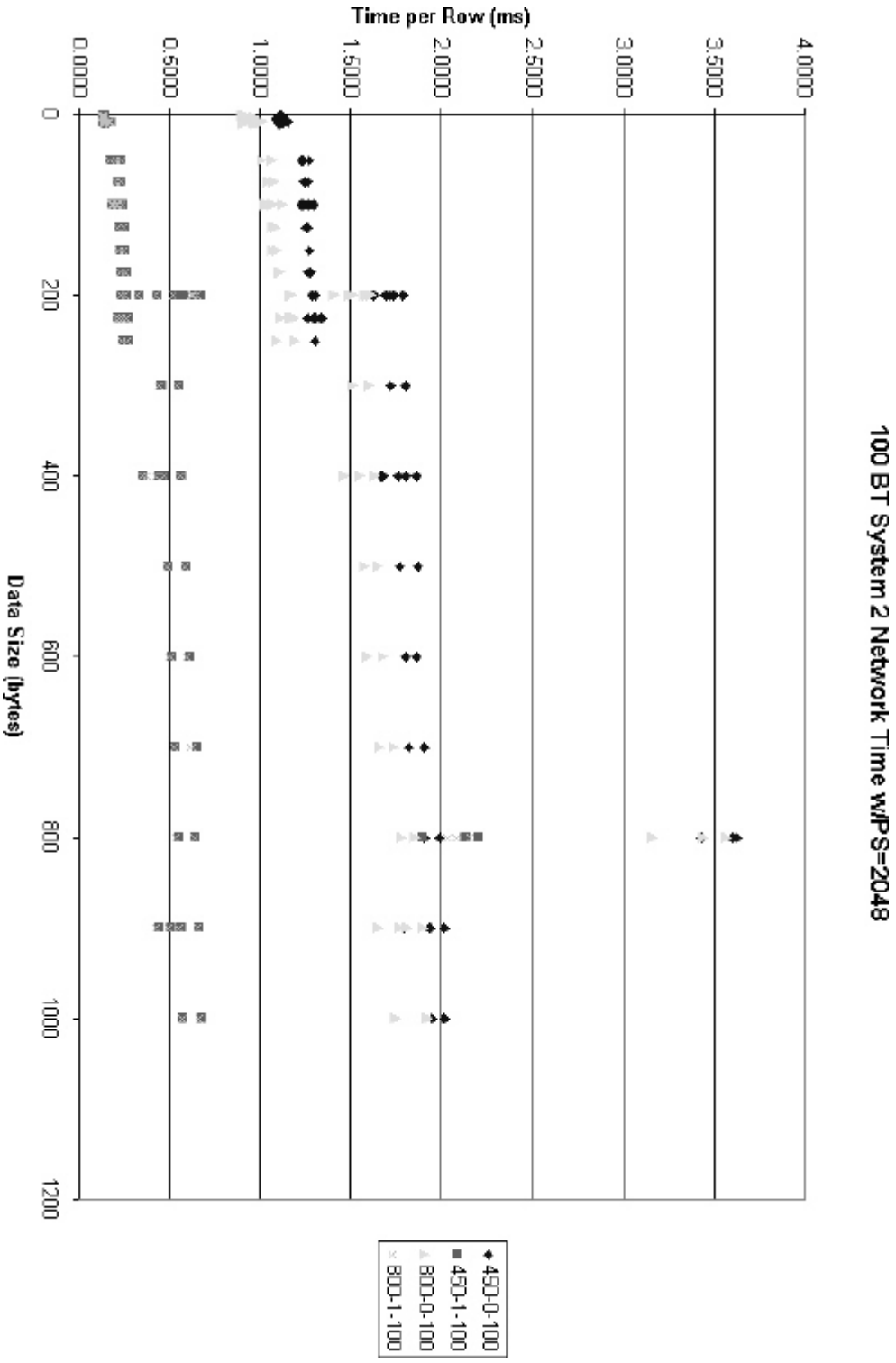

Figure 73: System 2 Network Time with a 100 Base-T link and a packet size of 2048 bytes.

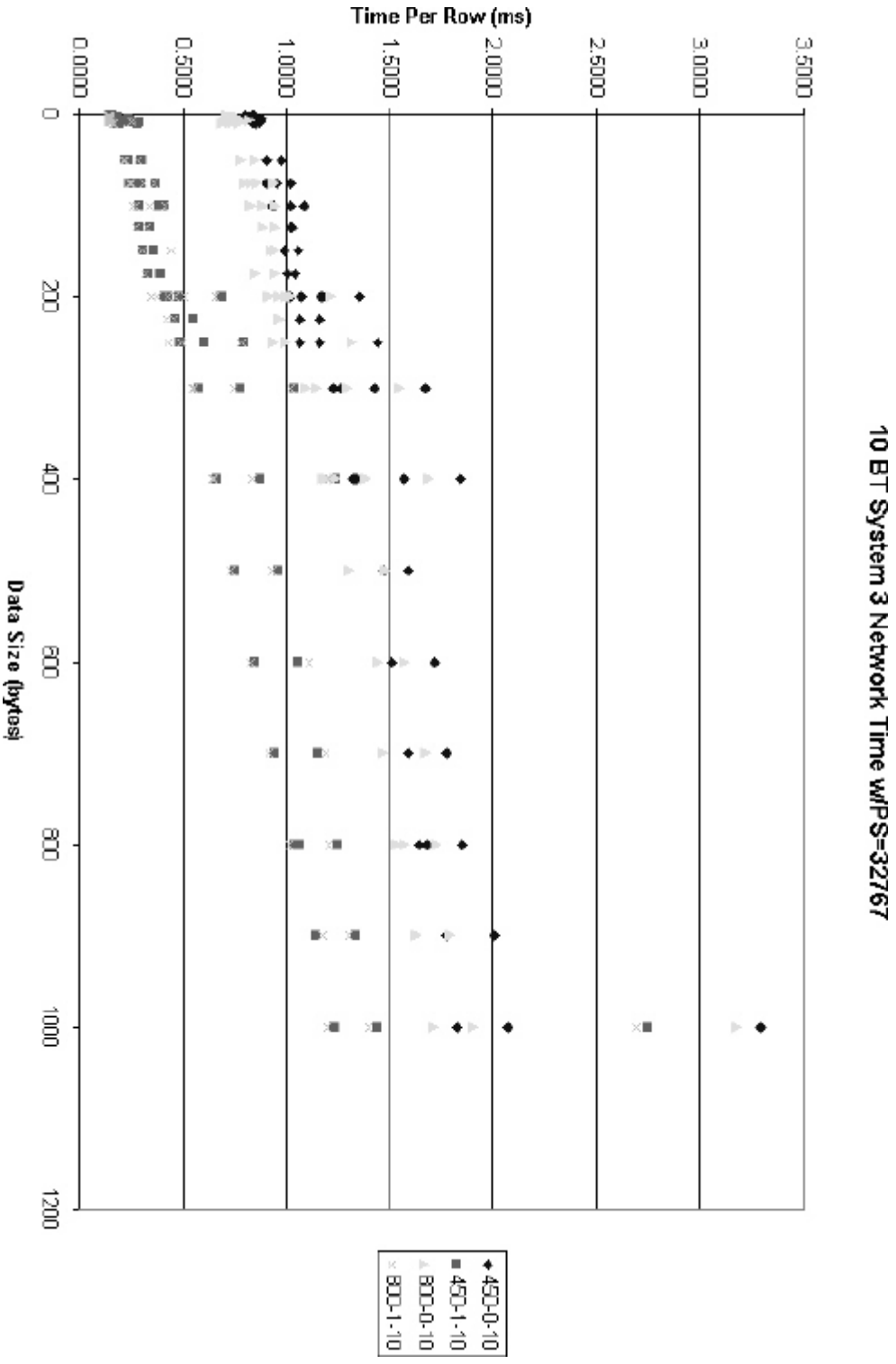

Figure 74: System 3 Network Time with a 10 Base-T link and a packet size of 32767 bytes.

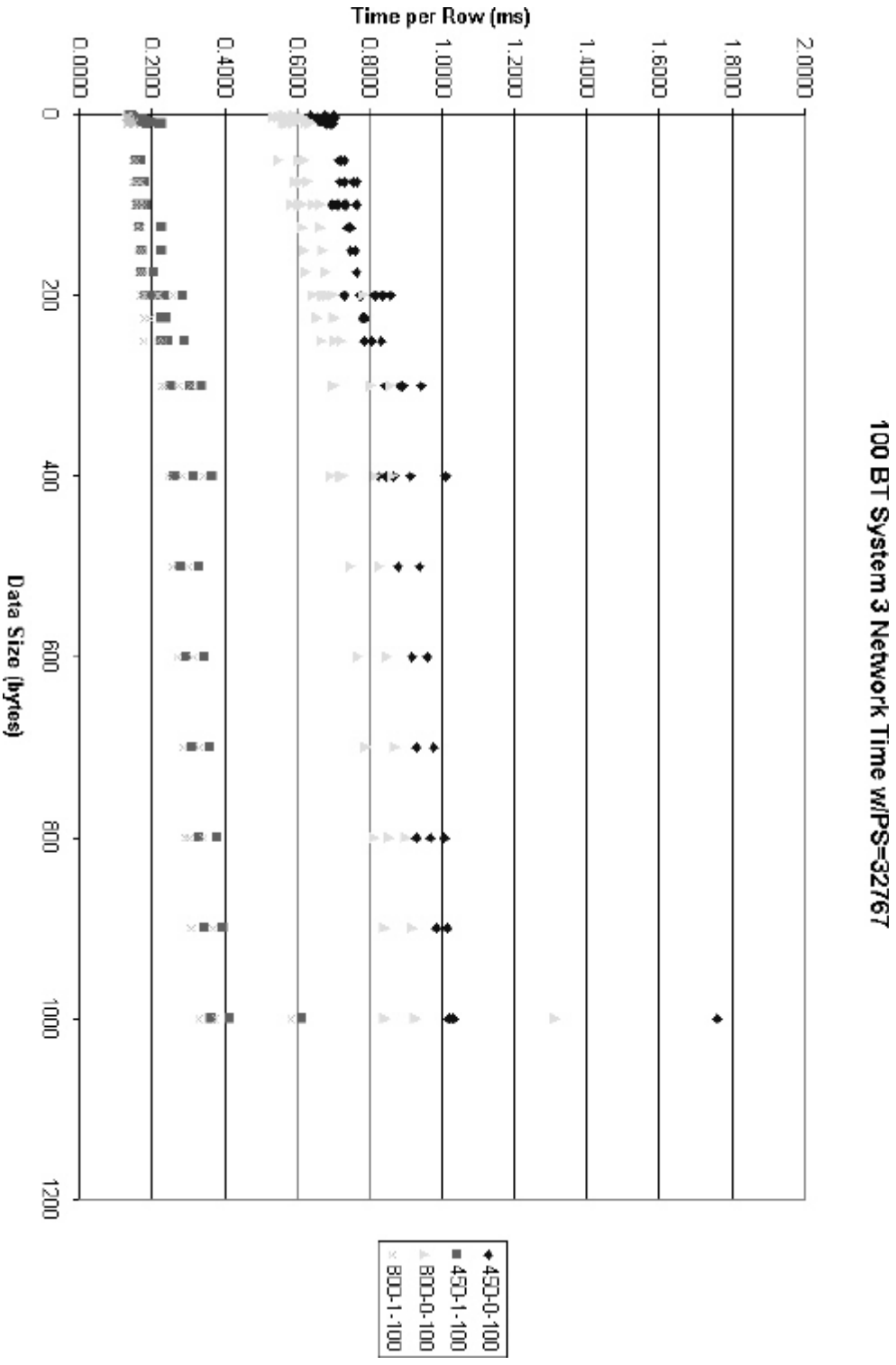

Figure 75: System 3 Network Time with a 100 Base-T link and a packet size of 32767 bytes.

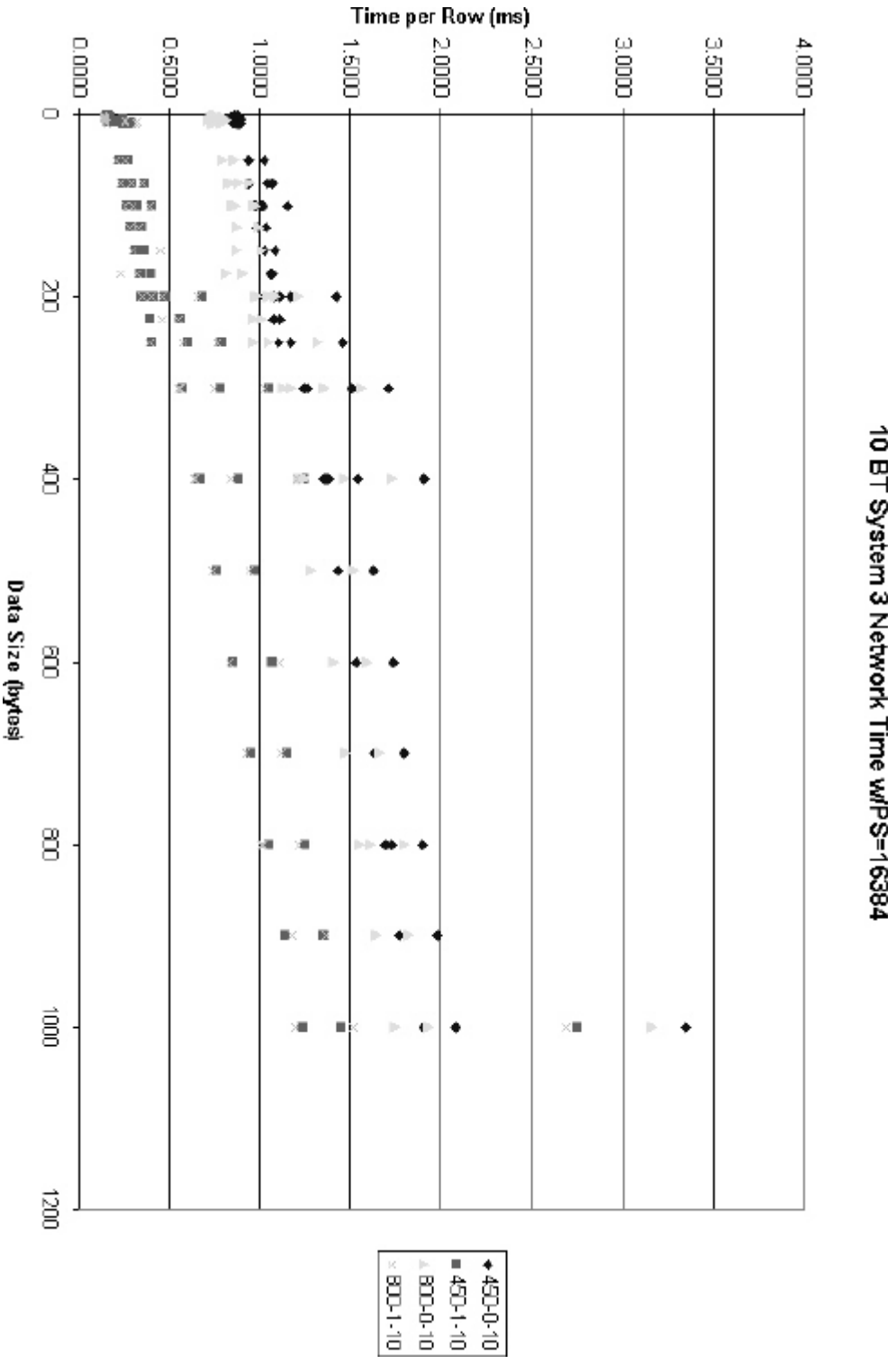

Figure 76: System 3 Network Time with a 10 Base-T link and a packet size of 16384 bytes.

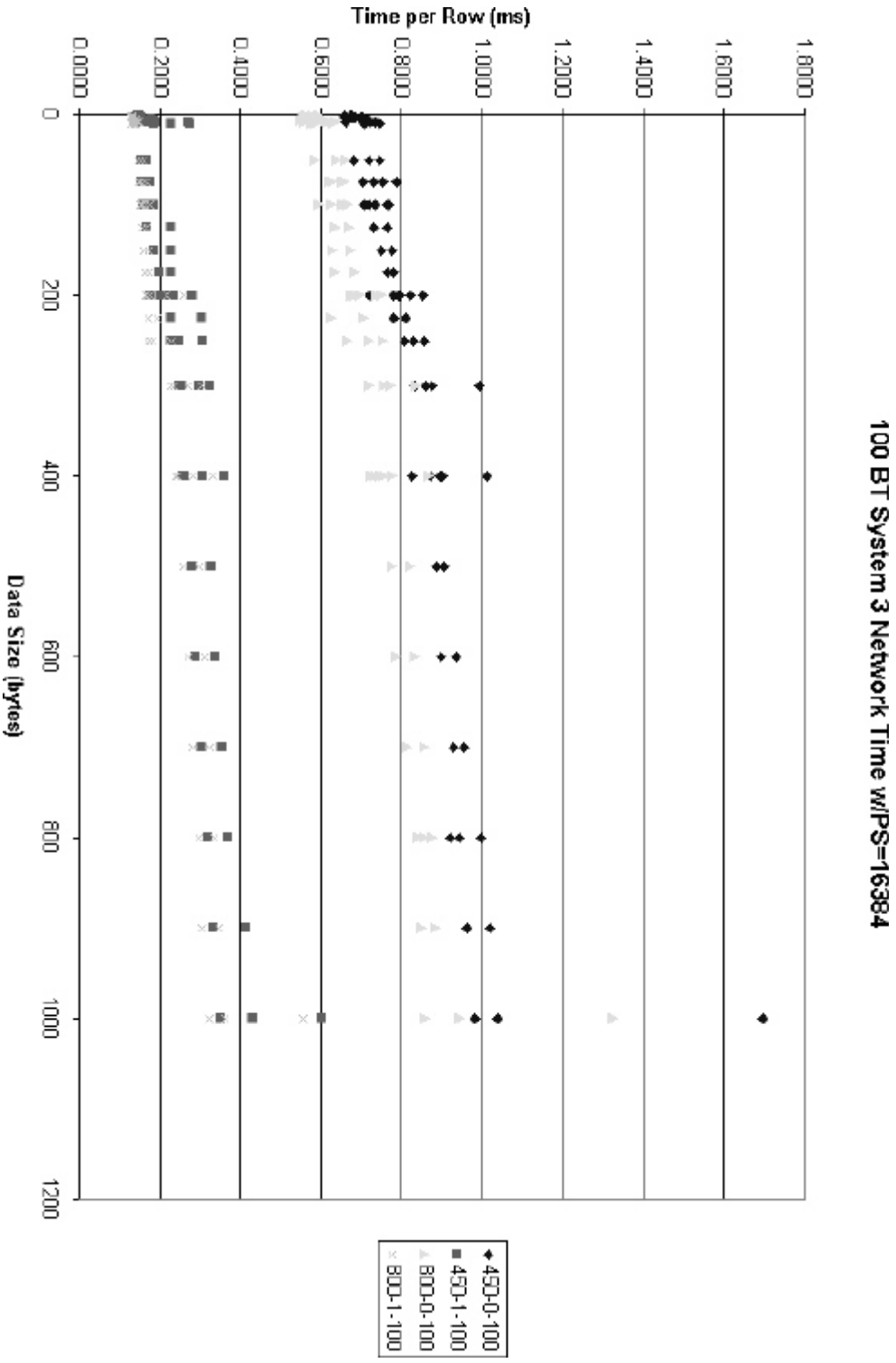

Figure 77: System 3 Network Time with a 100 Base-T link and a packet size of 16384 bytes.

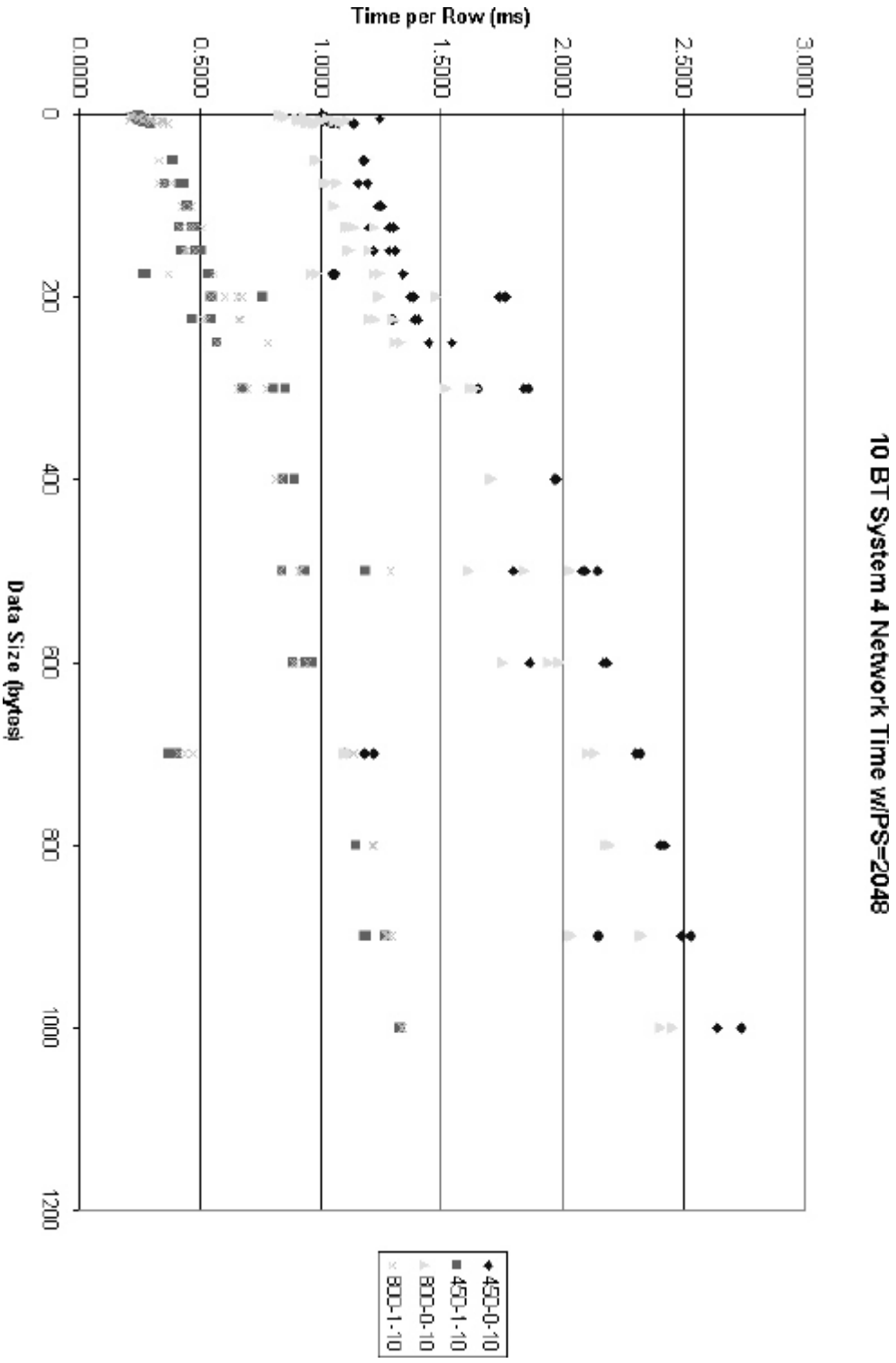

Figure 78: System 4 Network Time with a 10 Base-T link and a packet size of 2048 bytes.

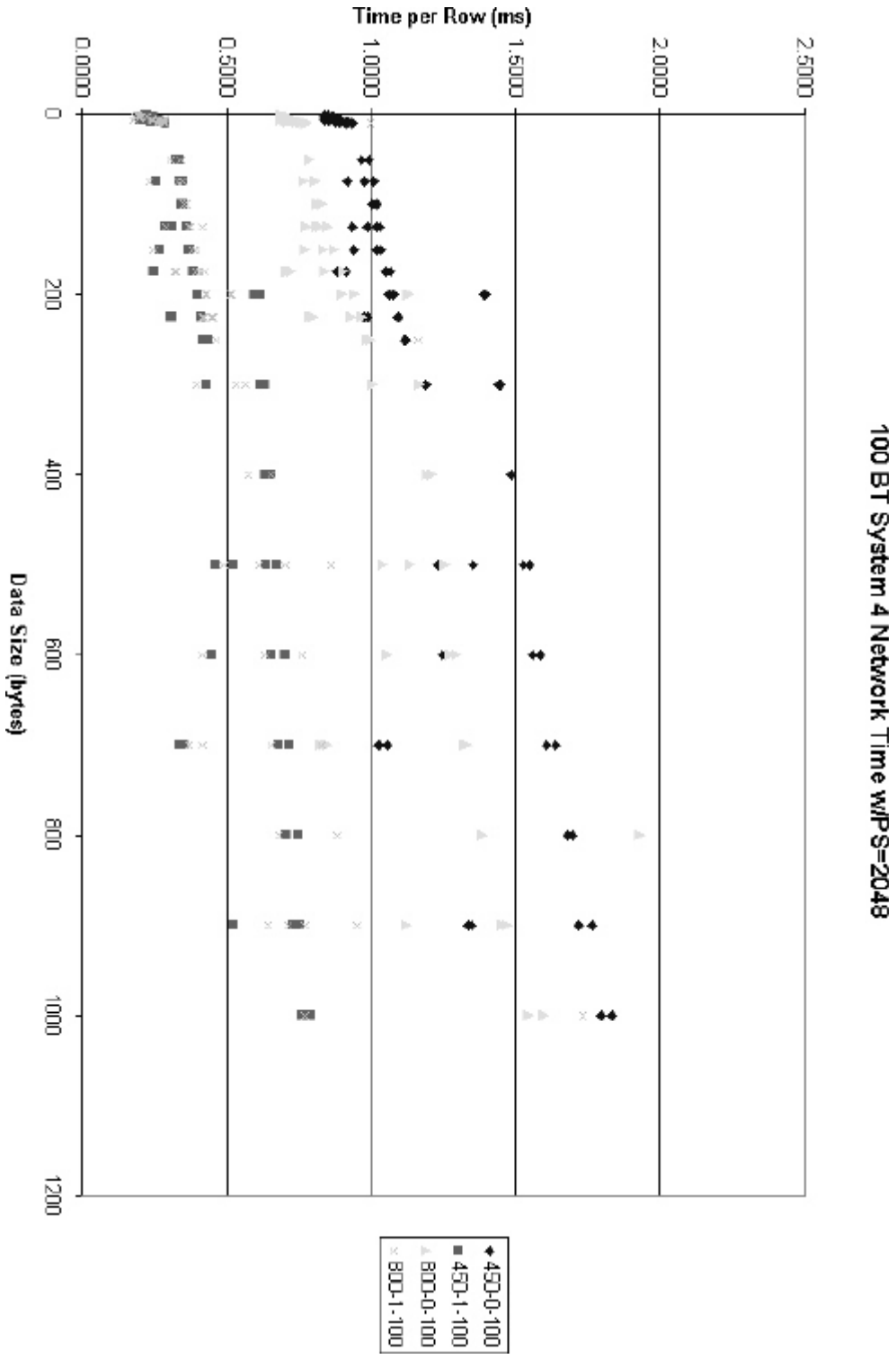

Figure 79: System 4 Network Time with a 100 Base-T link and a packet size of 2048 bytes.

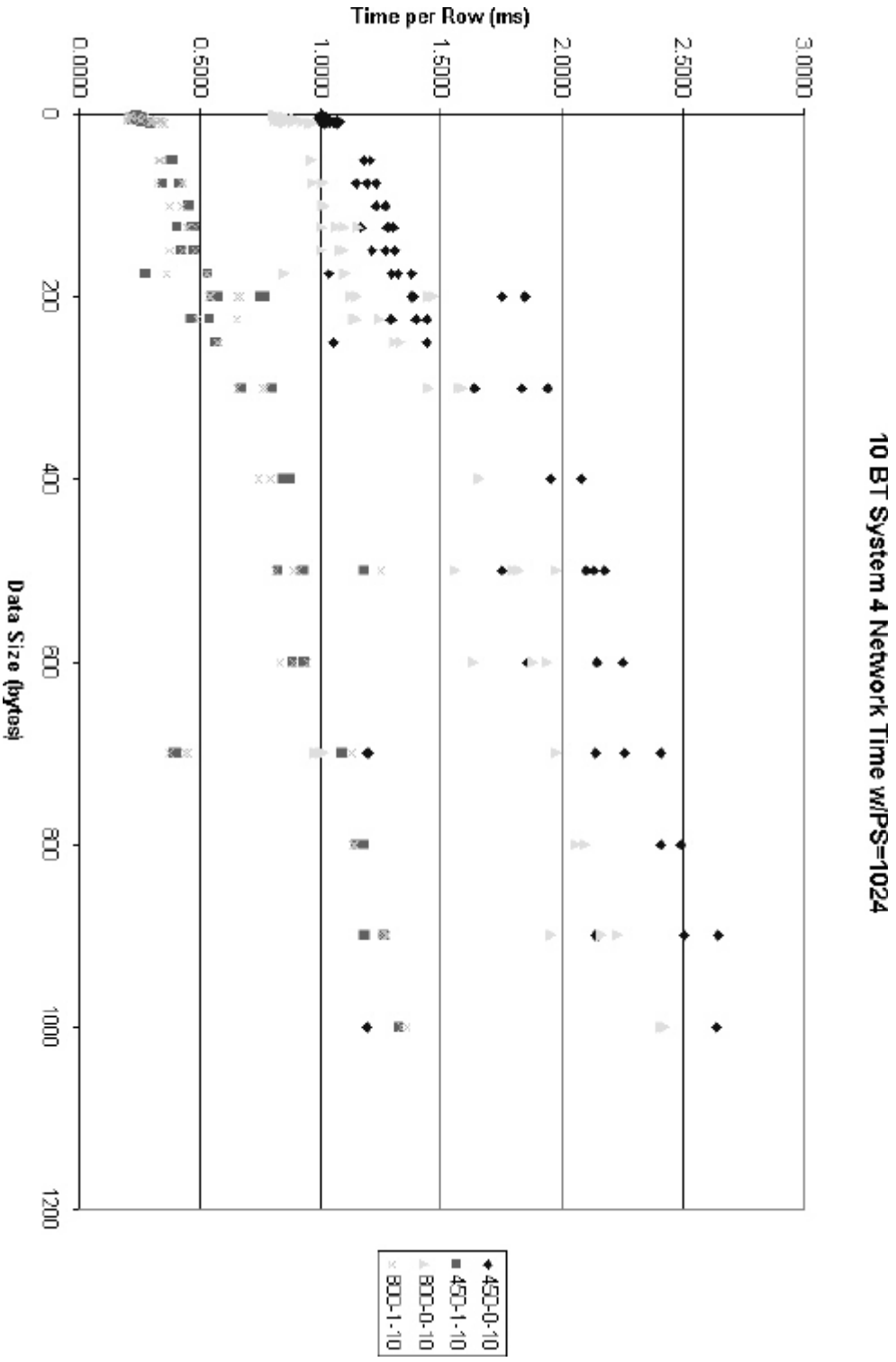

Figure 80: System 4 Network Time with a 10 Base-T link and a packet size of 1024 bytes.

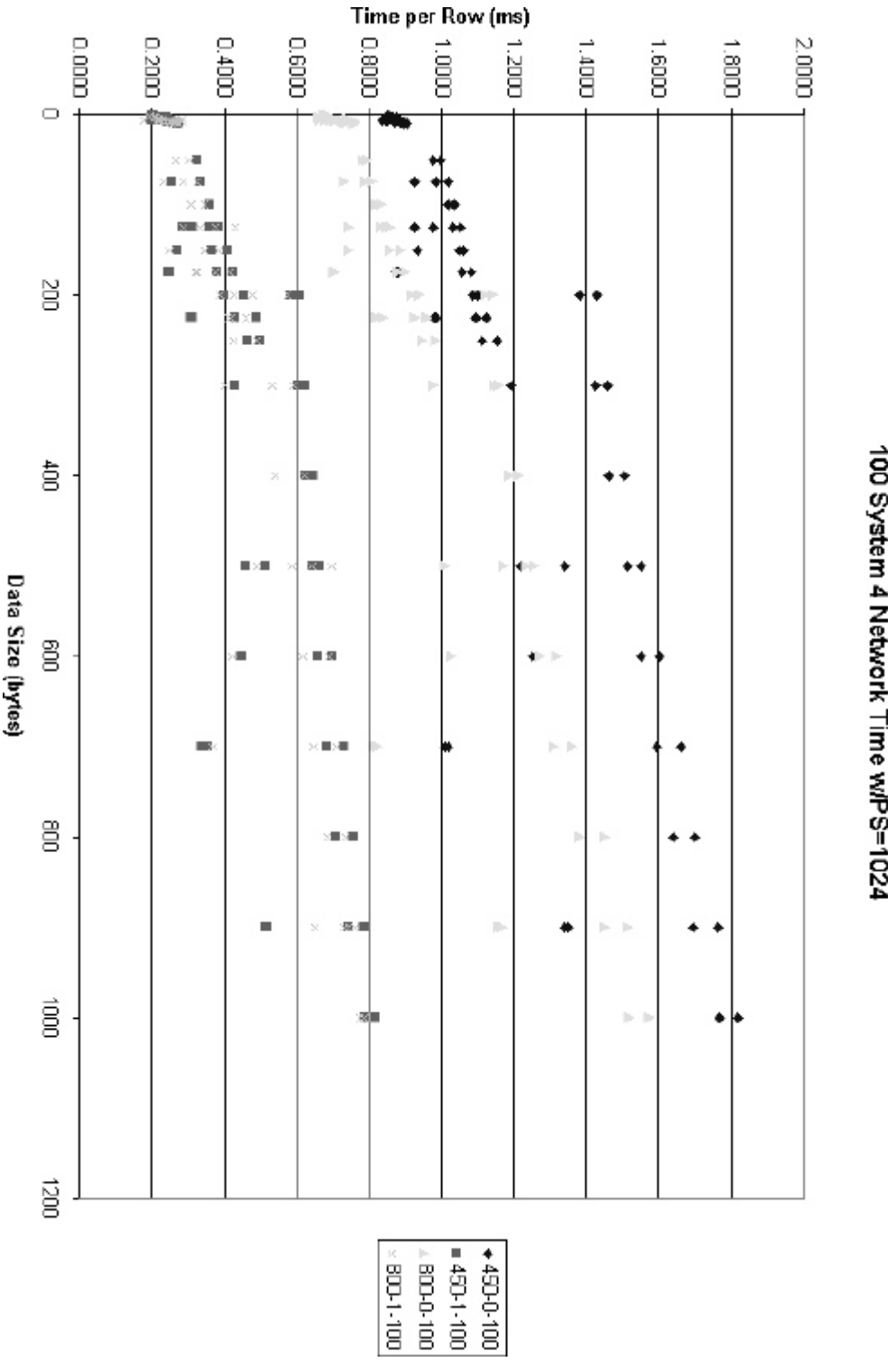

Figure 81: System 4 Network Time with a 100 Base-T link and a packet size of 1024 bytes.

D.2 Bar Charts

This section contains those bar charts referenced in section 6.

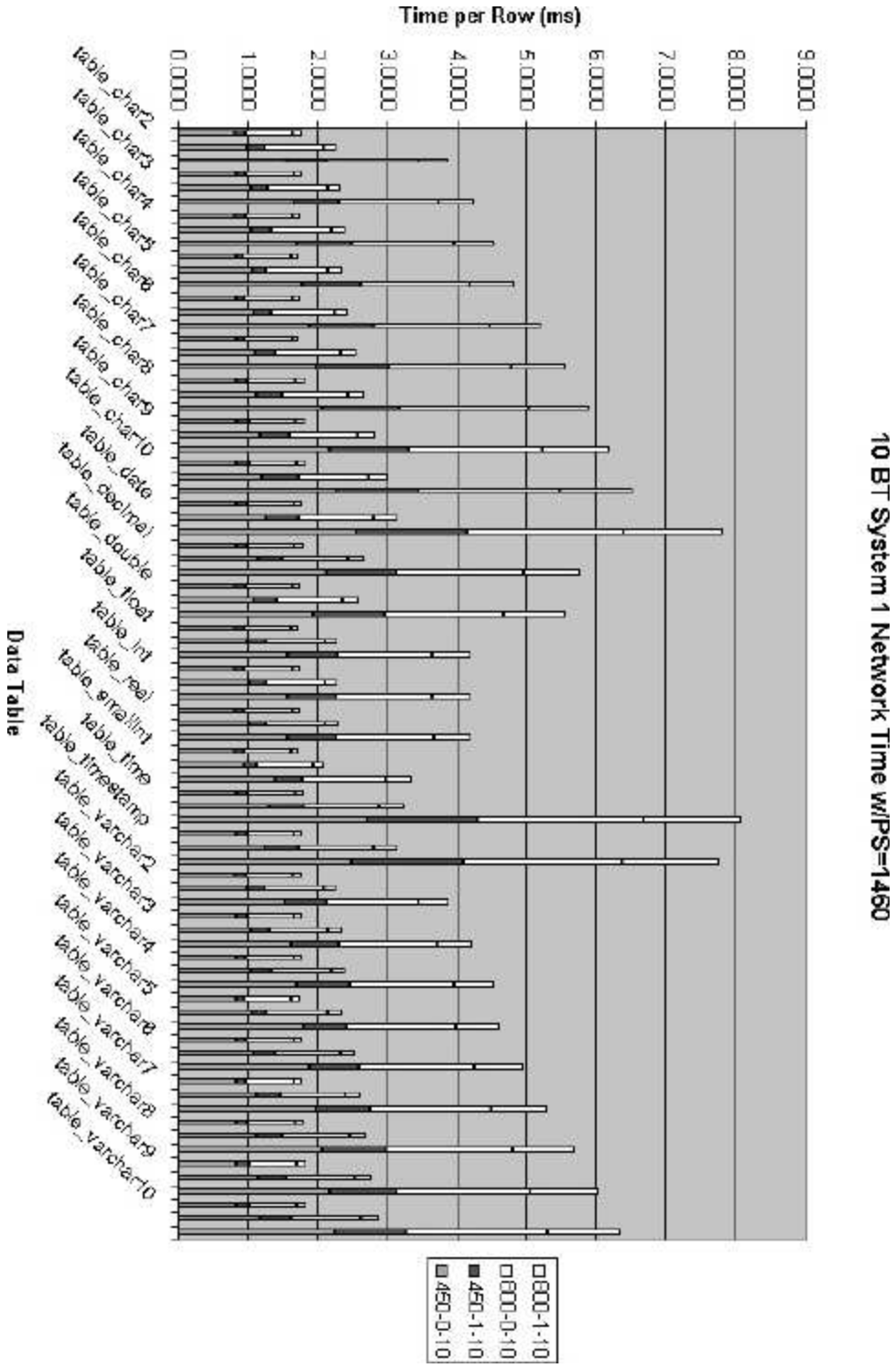

Figure 82: System 1 Network Time with a 10 Base-T link and a packet size of 1460 bytes.

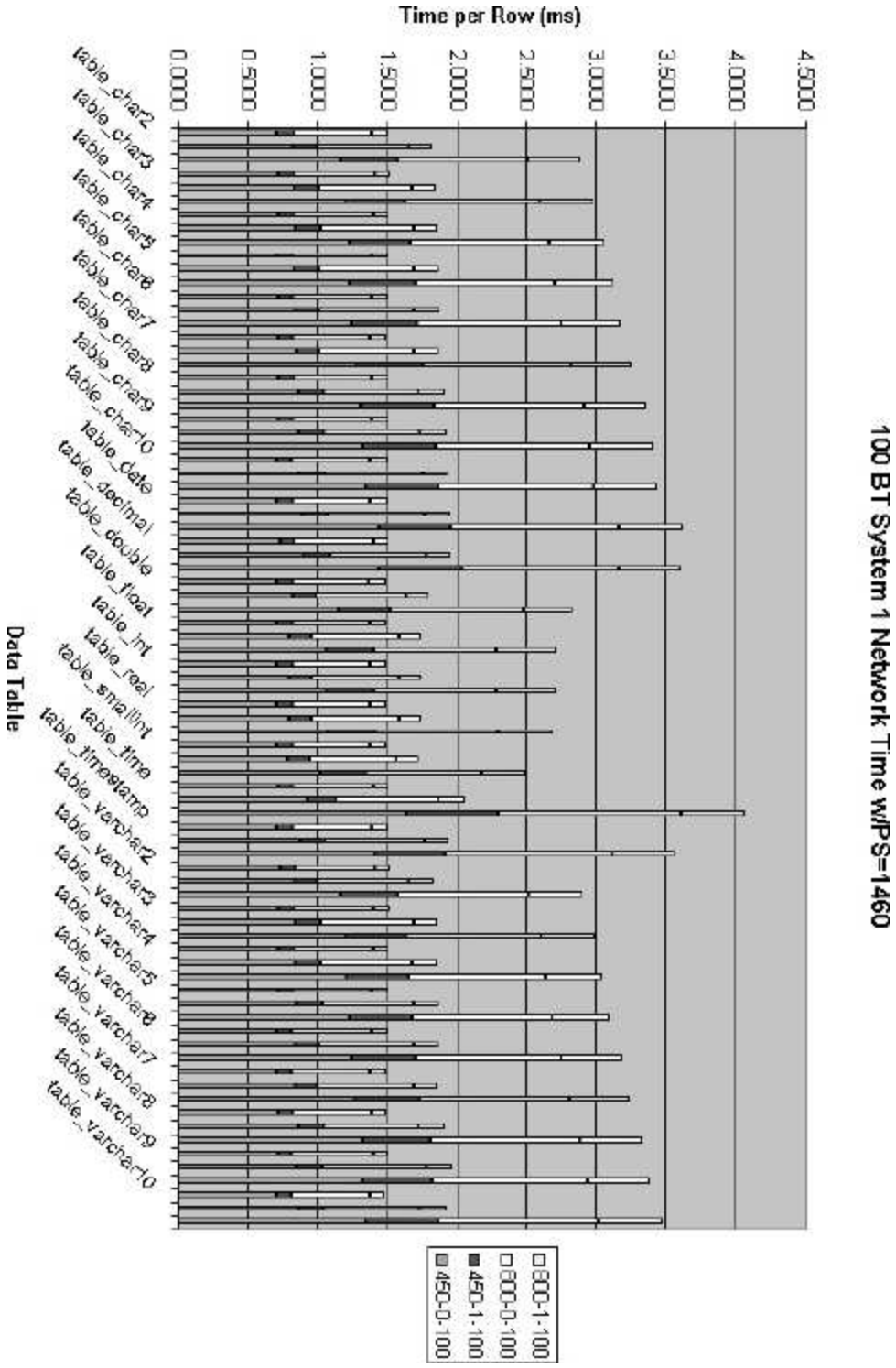

Figure 83: System 1 Network Time with a 100 Base-T link and a packet size of 1460 bytes.

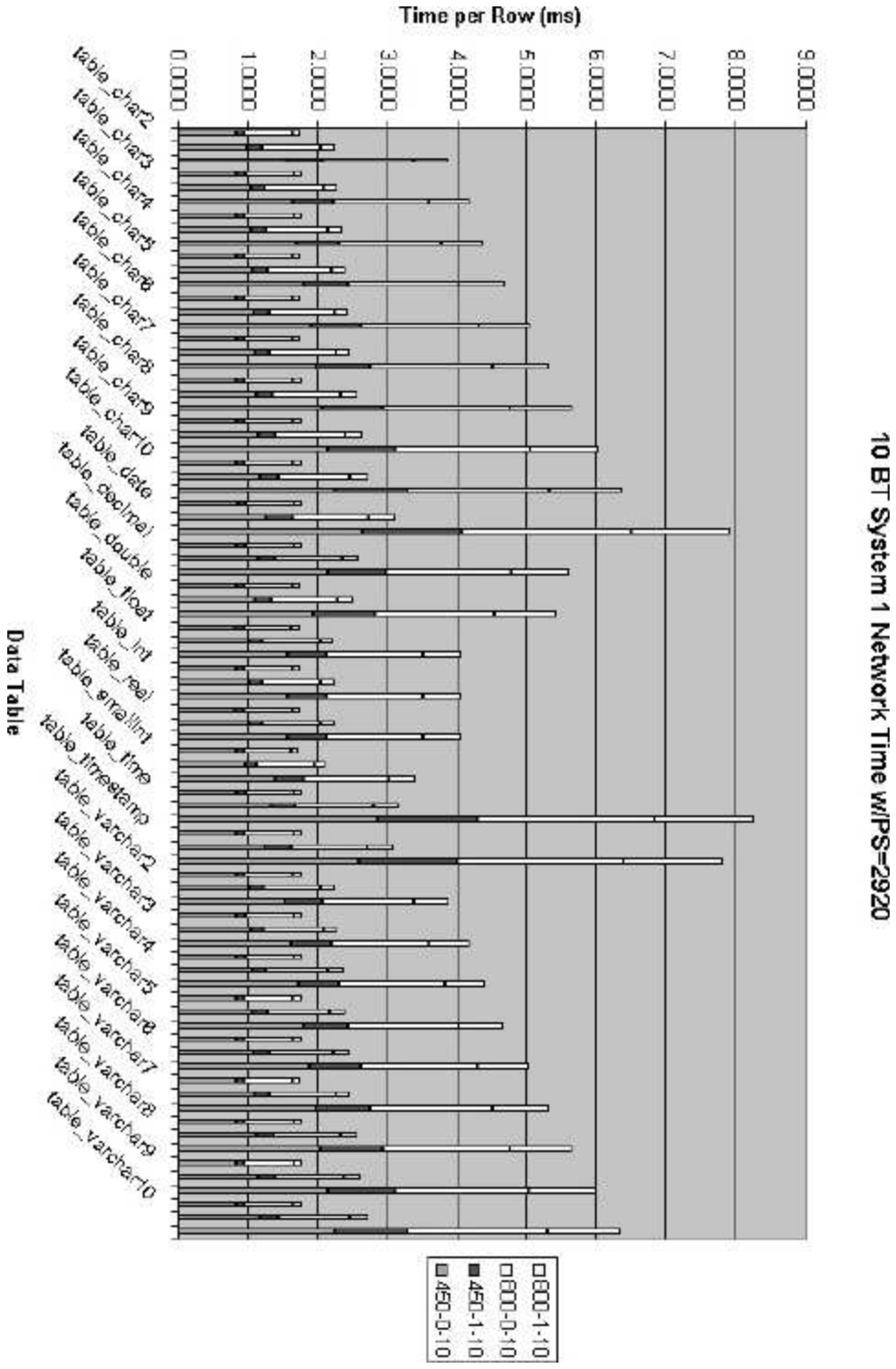

Figure 84: System 1 Network Time with a 10 Base-T link and a packet size of 2920 bytes.

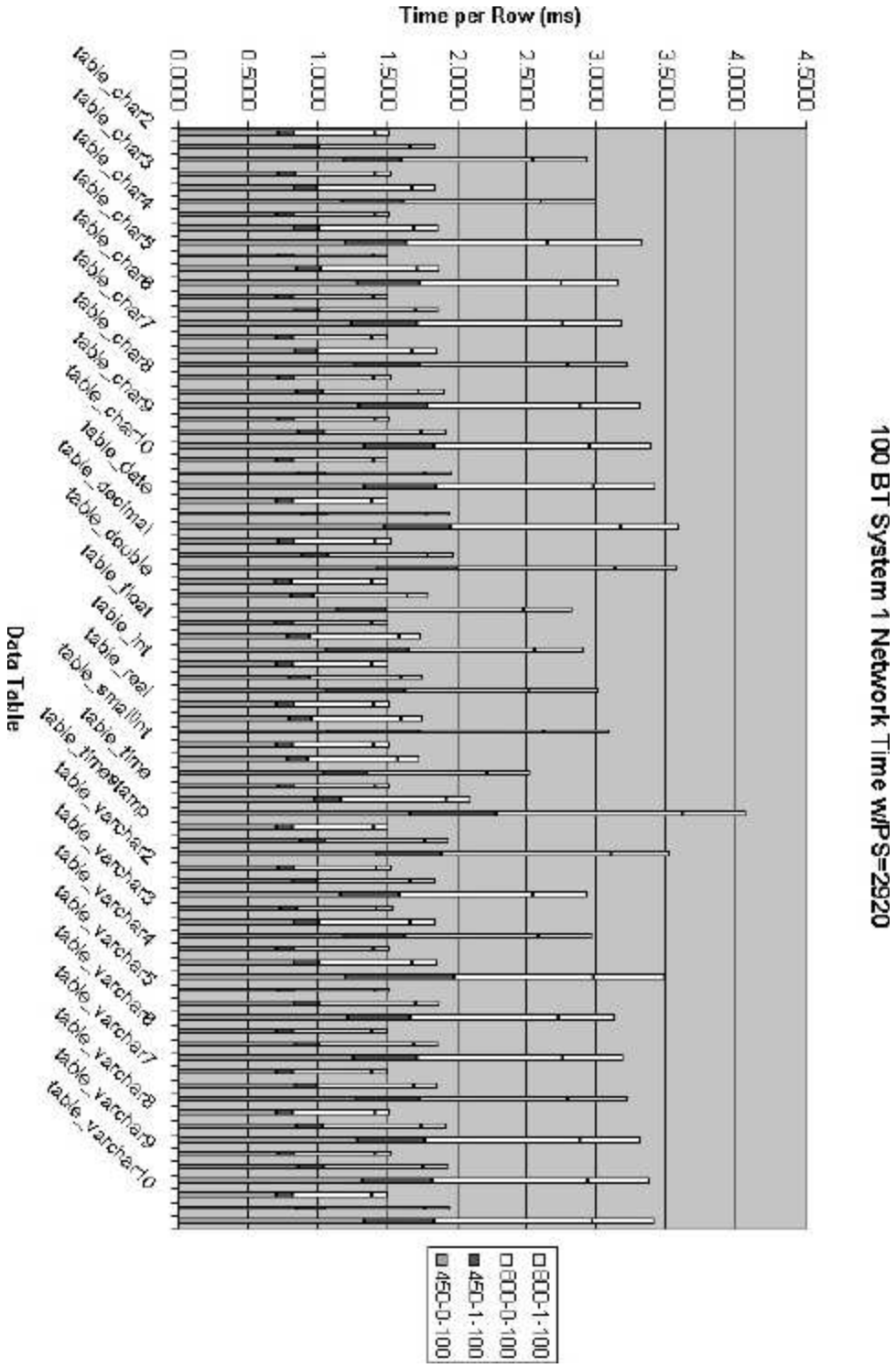

Figure 85: System 1 Network Time with a 100 Base-T link and a packet size of 2920 bytes.

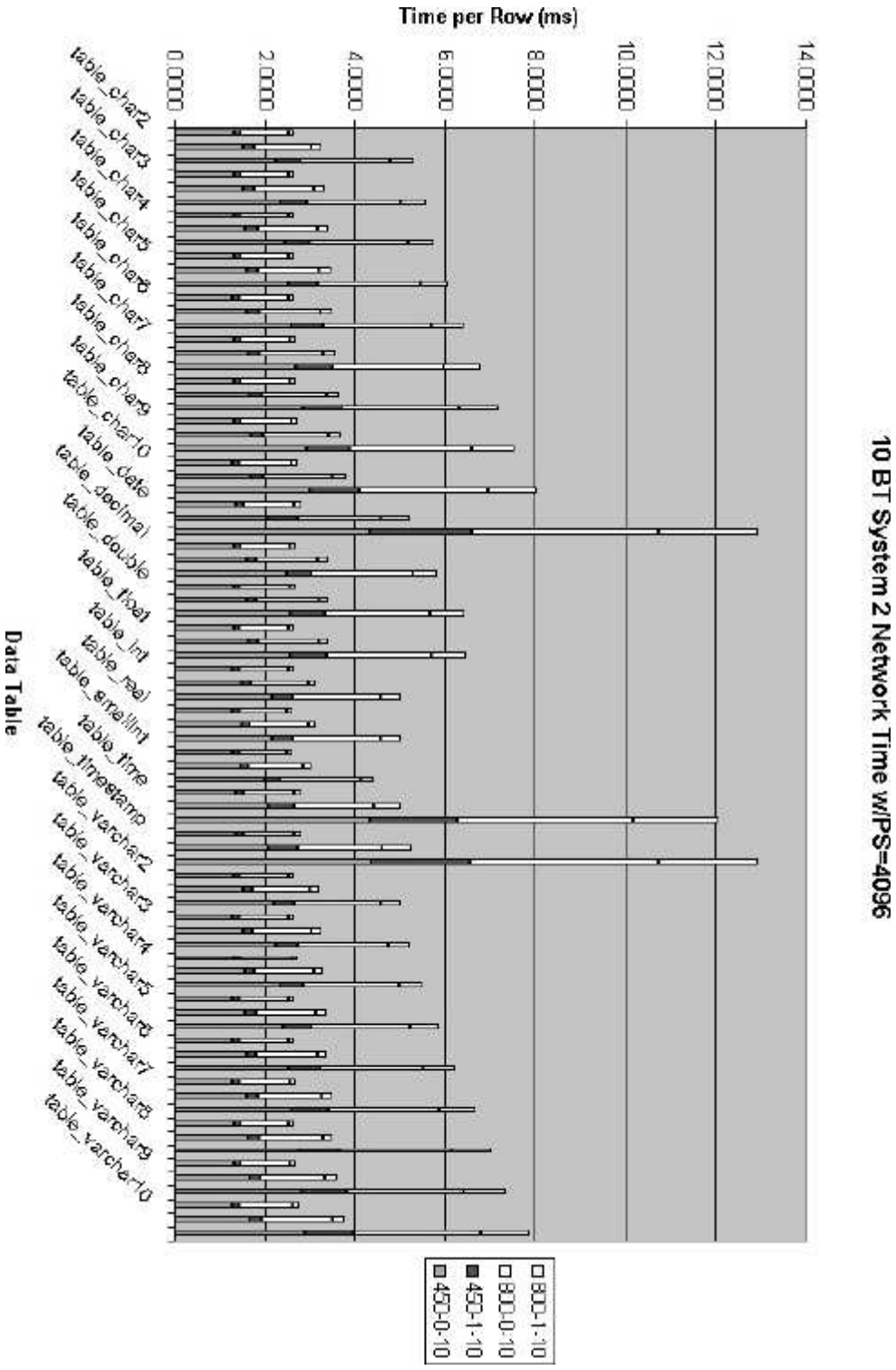

Figure 86: System 2 Network Time with a 10 Base-T link and a packet size of 4096 bytes.

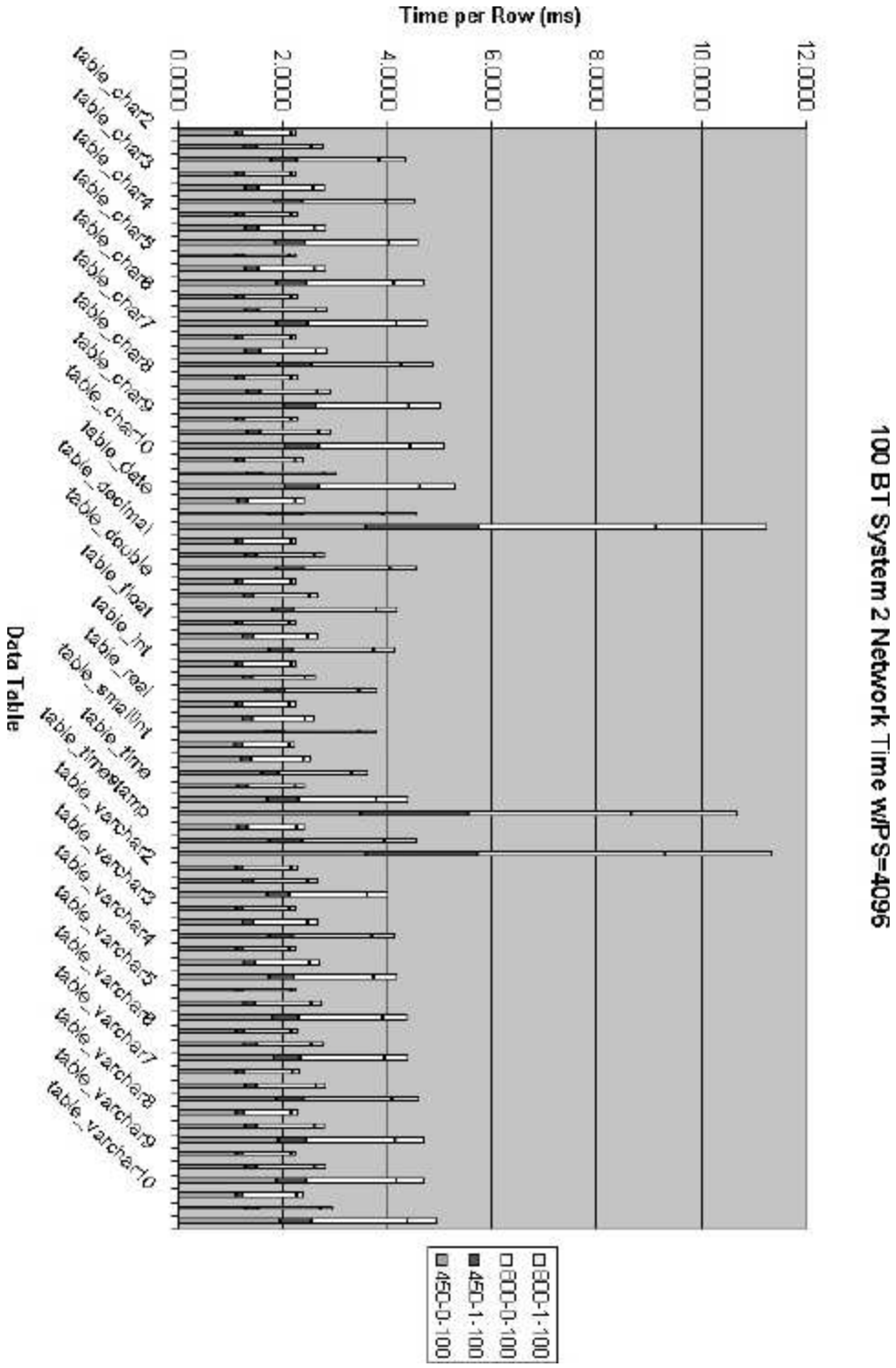

Figure 87: System 2 Network Time with a 100 Base-T link and a packet size of 4096 bytes.

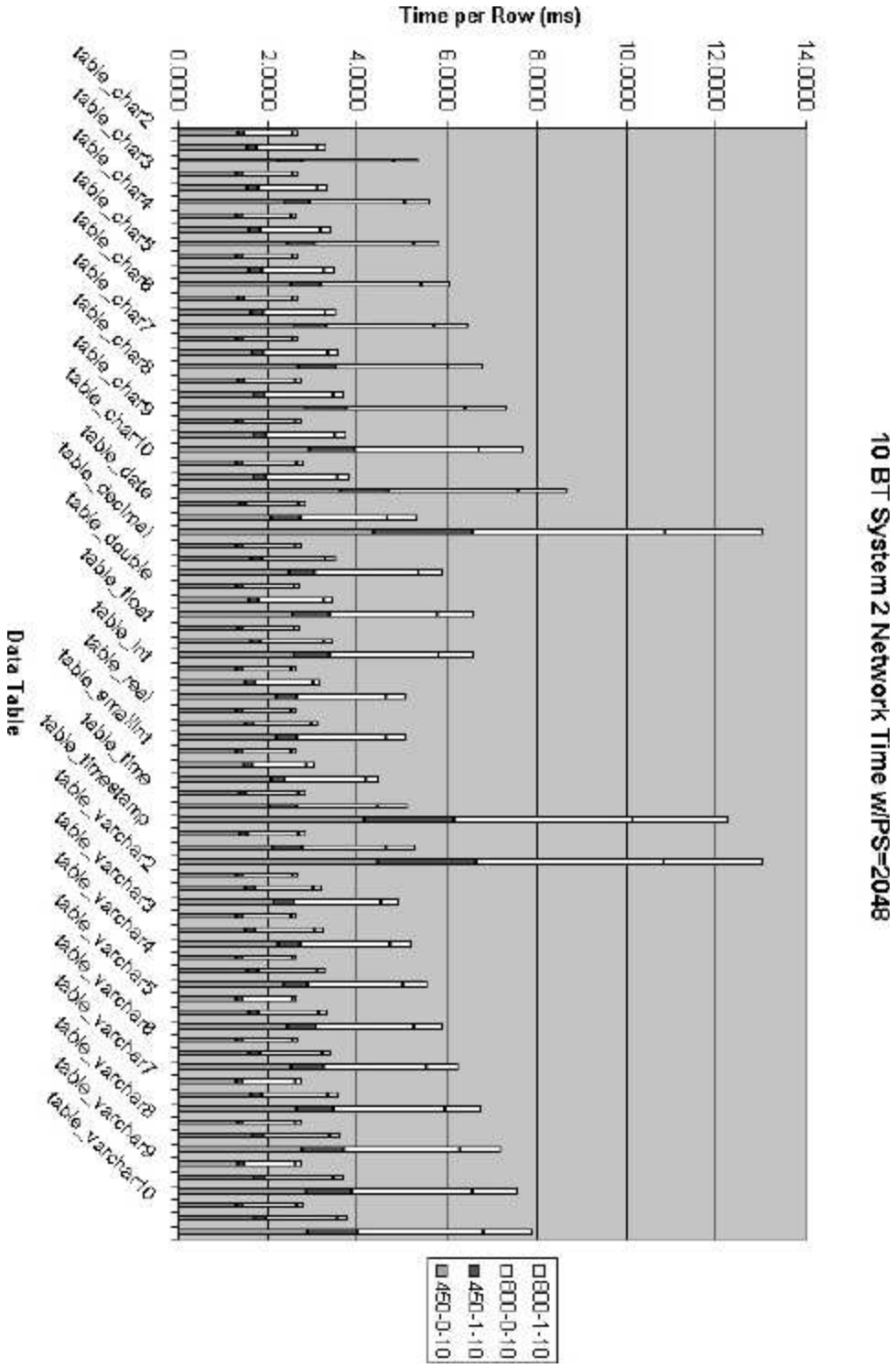

Figure 88: System 2 Network Time with a 10 Base-T link and a packet size of 2048 bytes.

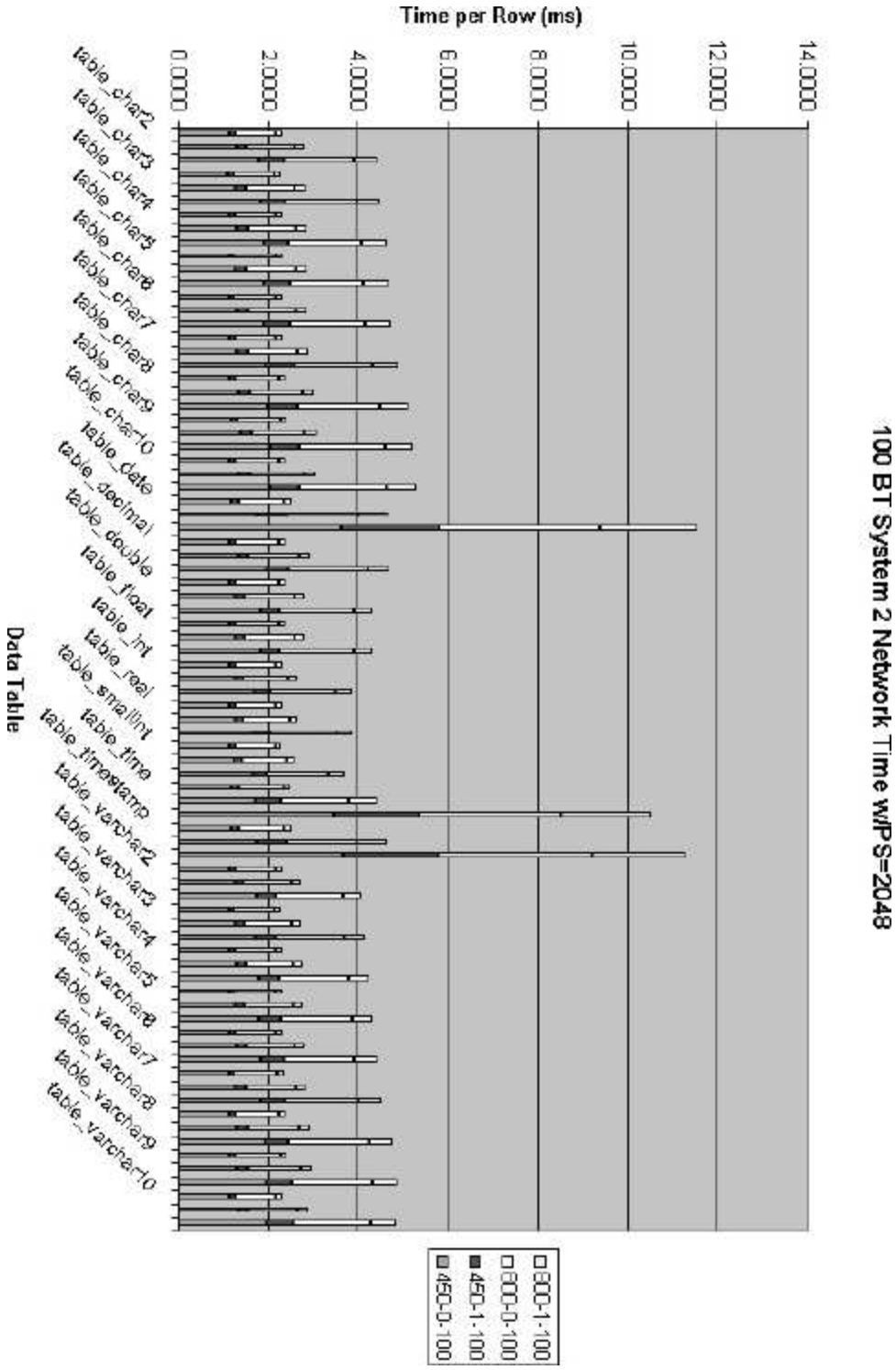

Figure 89: System 2 Network Time with a 100 Base-T link and a packet size of 2048 bytes.

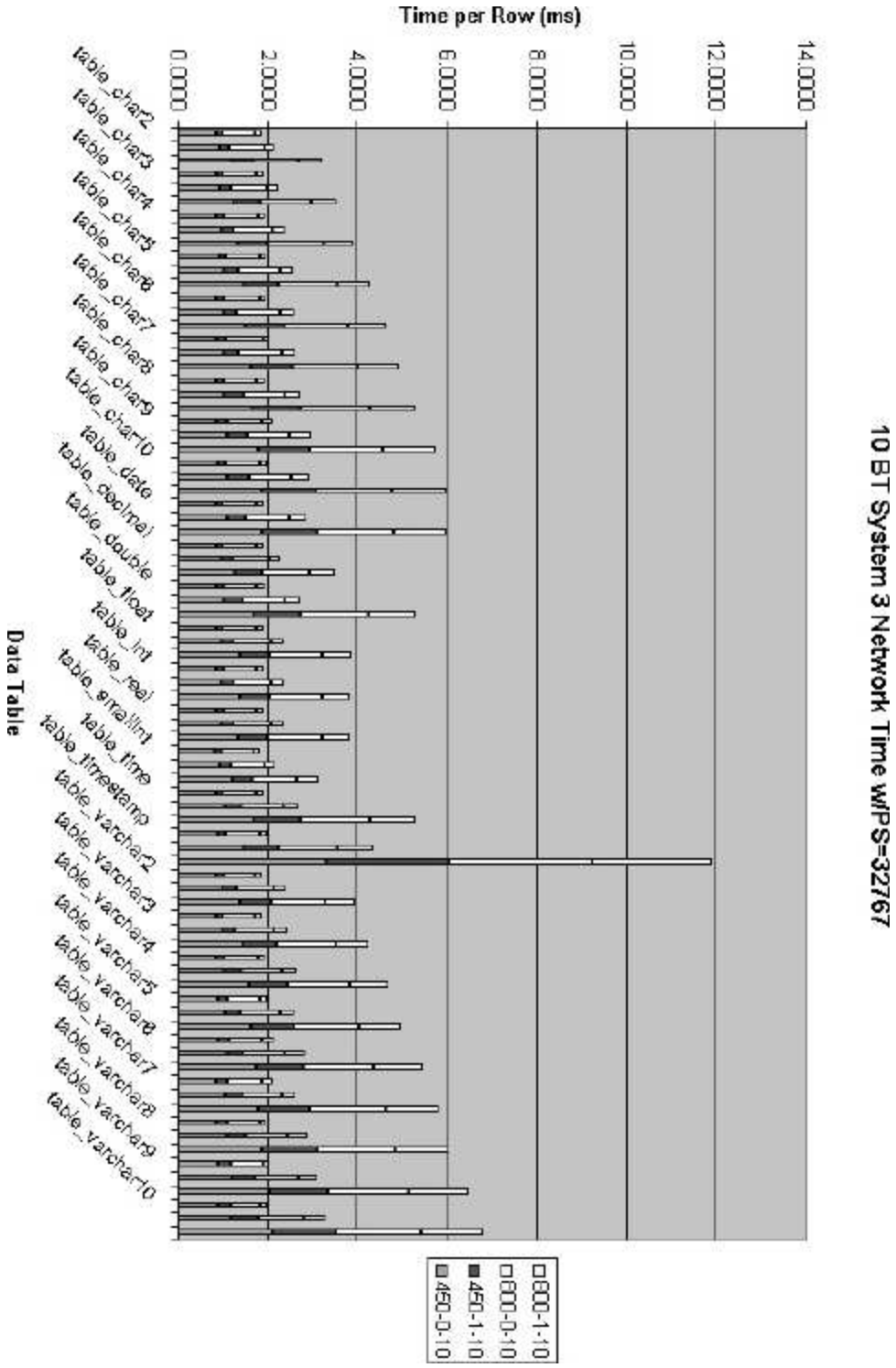

Figure 90: System 3 Network Time with a 10 Base-T link and a packet size of 32767 bytes.

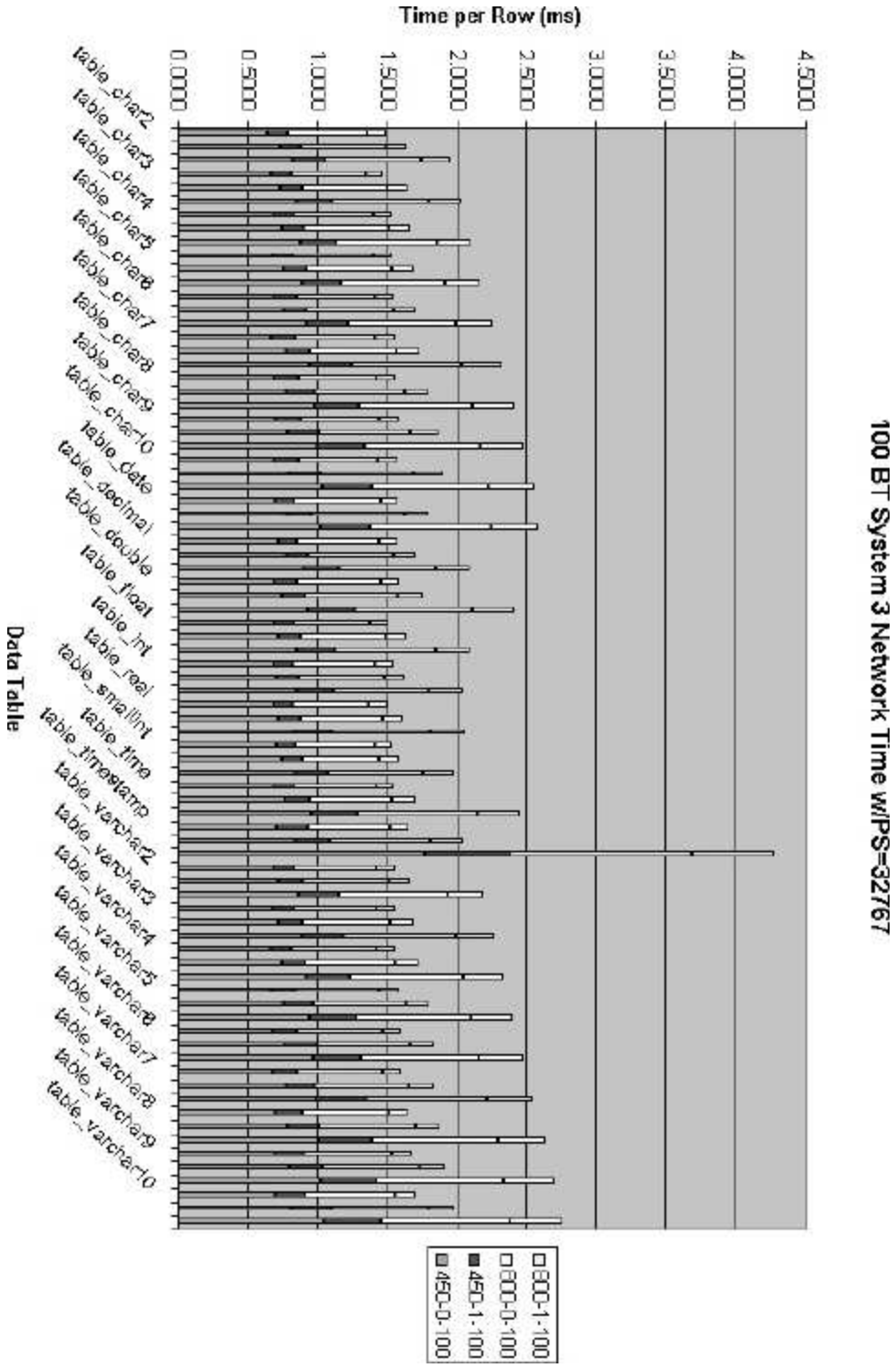

Figure 91: System 3 Network Time with a 100 Base-T link and a packet size of 32767 bytes.

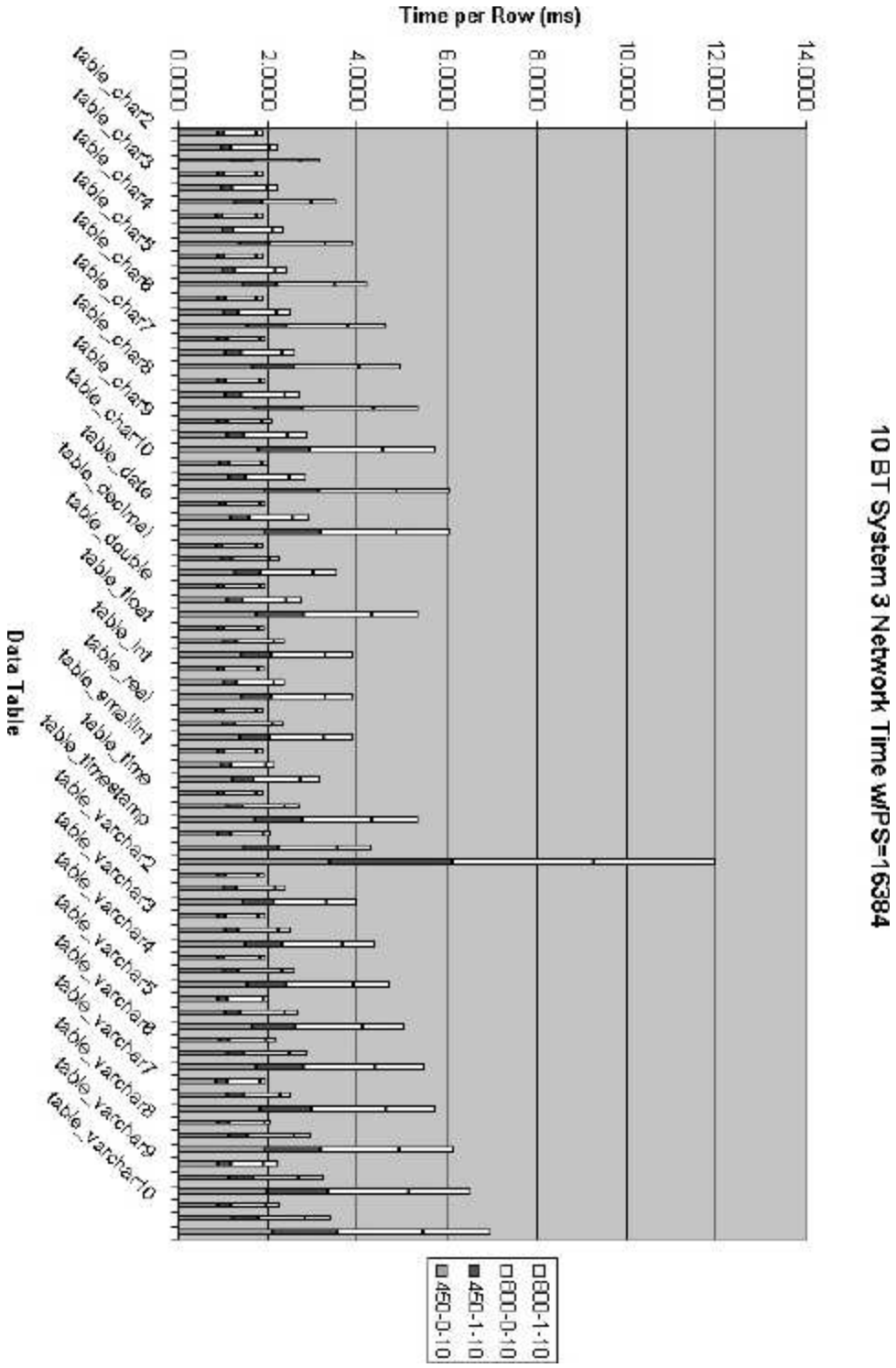

Figure 92: System 3 Network Time with a 10 Base-T link and a packet size of 16384 bytes.

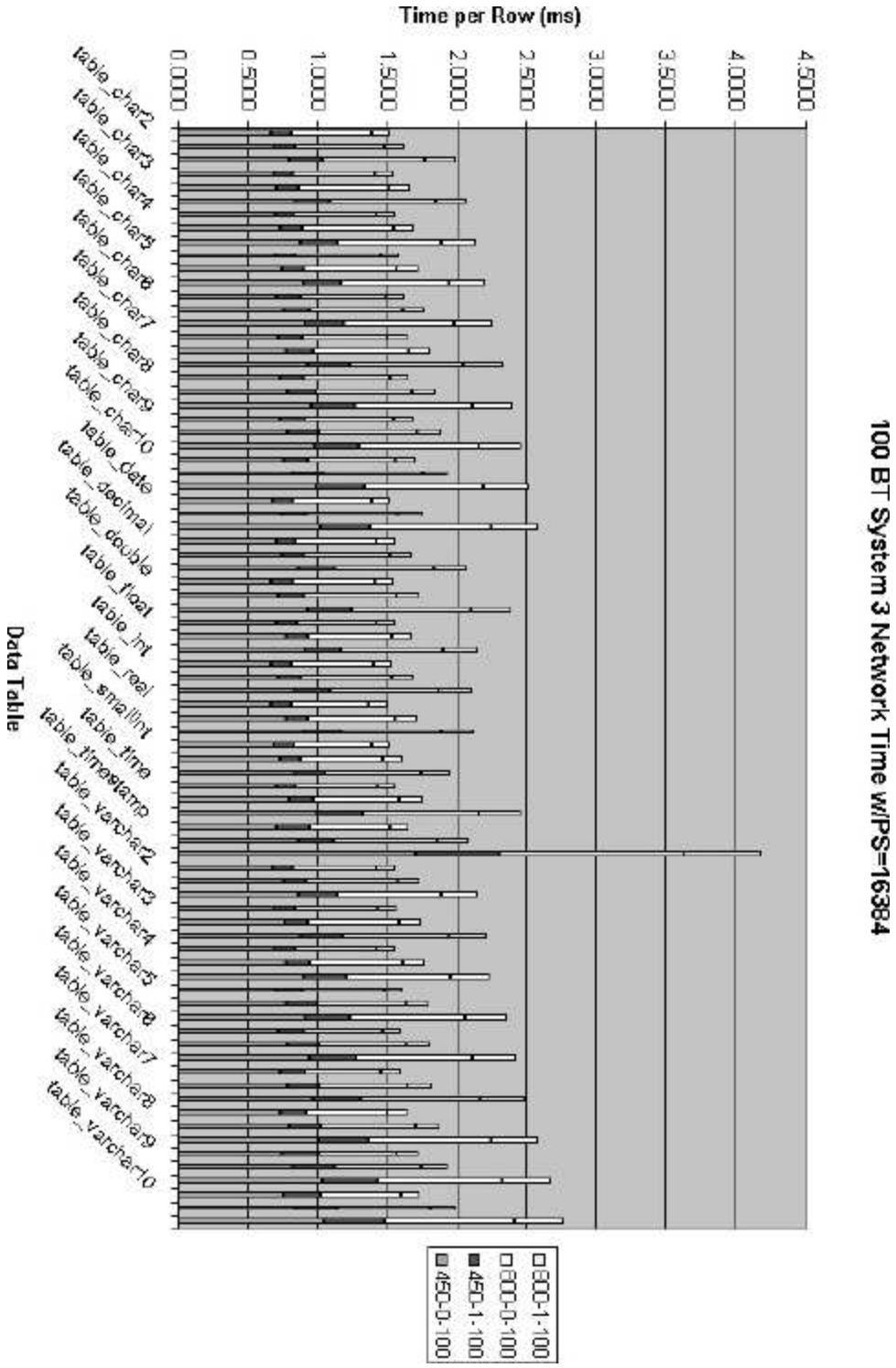

Figure 93: System 3 Network Time with a 100 Base-T link and a packet size of 16384 bytes.

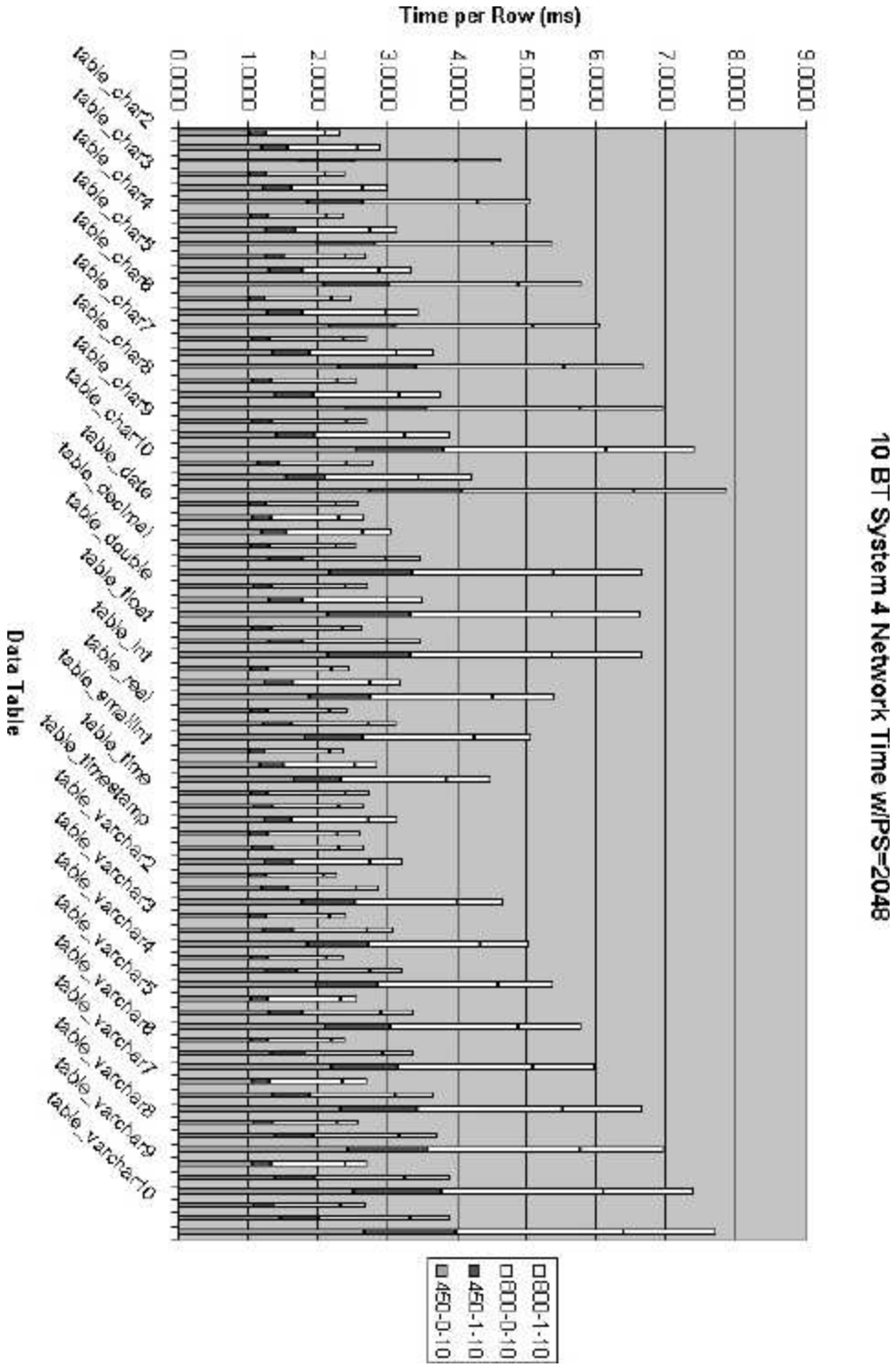

Figure 94: System 4 Network Time with a 10 Base-T link and a packet size of 2048 bytes.

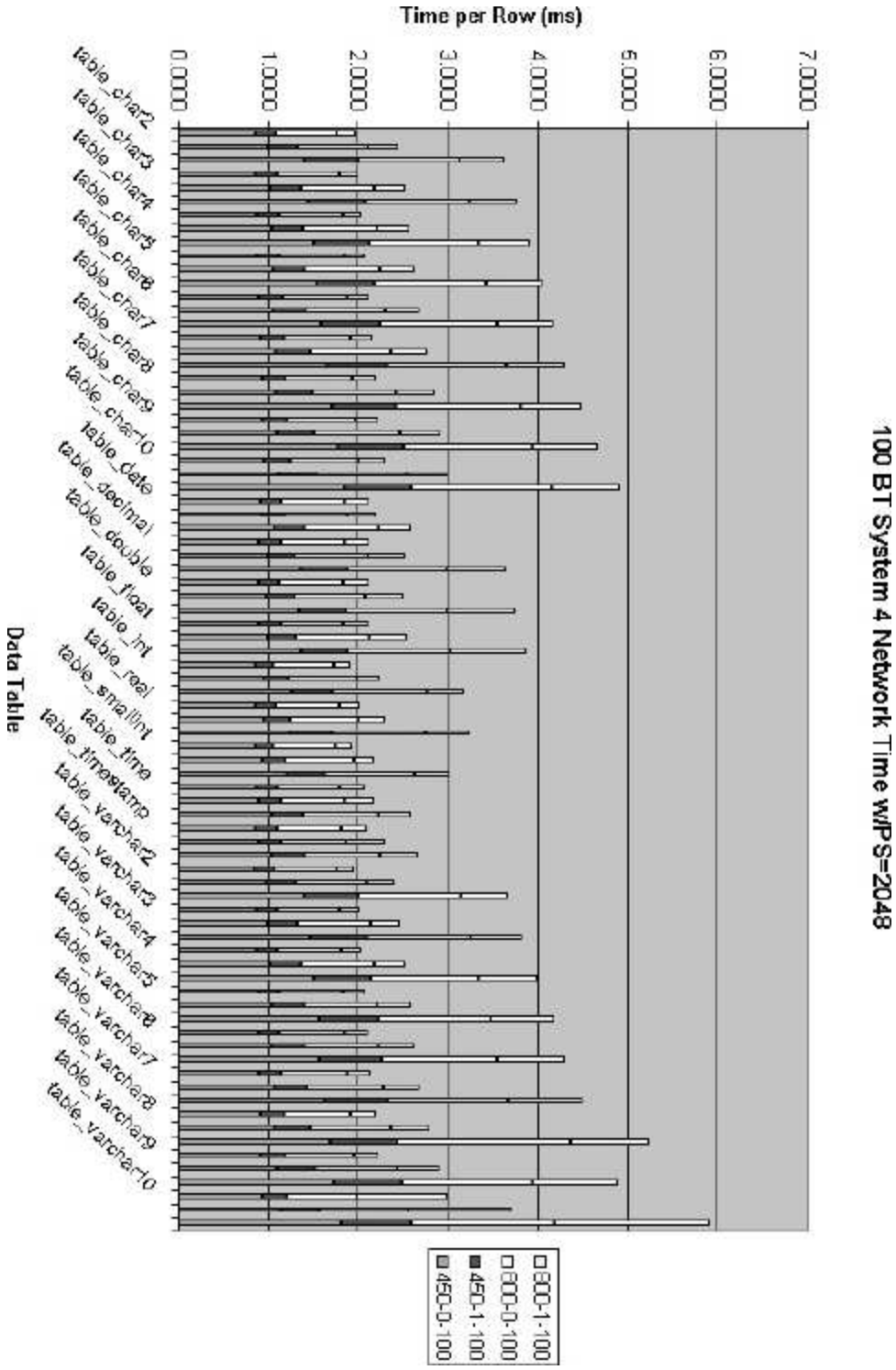

Figure 95: System 4 Network Time with a 100 Base-T link and a packet size of 2048 bytes.

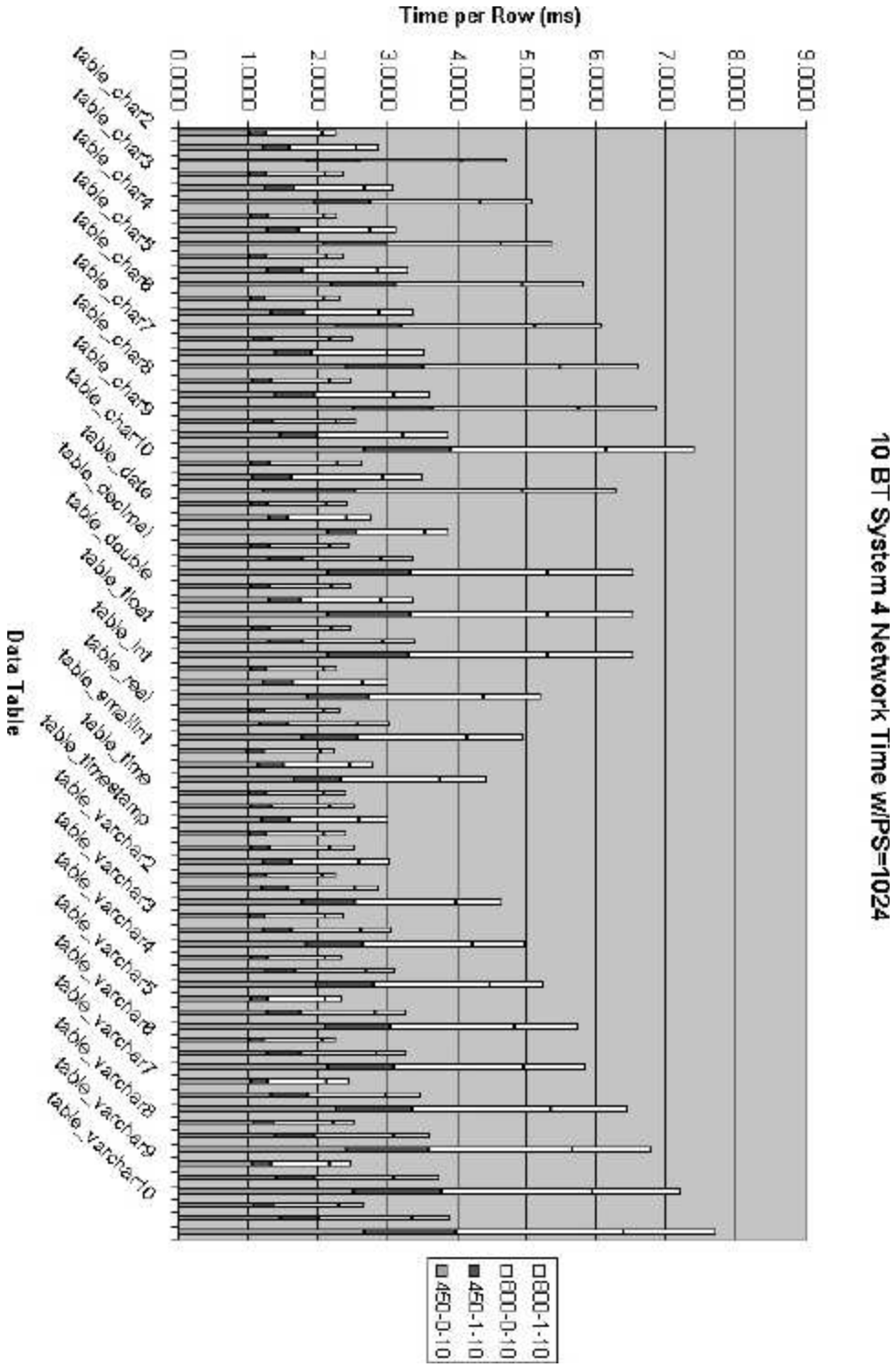

Figure 96: System 4 Network Time with a 10 Base-T link and a packet size of 1024 bytes.

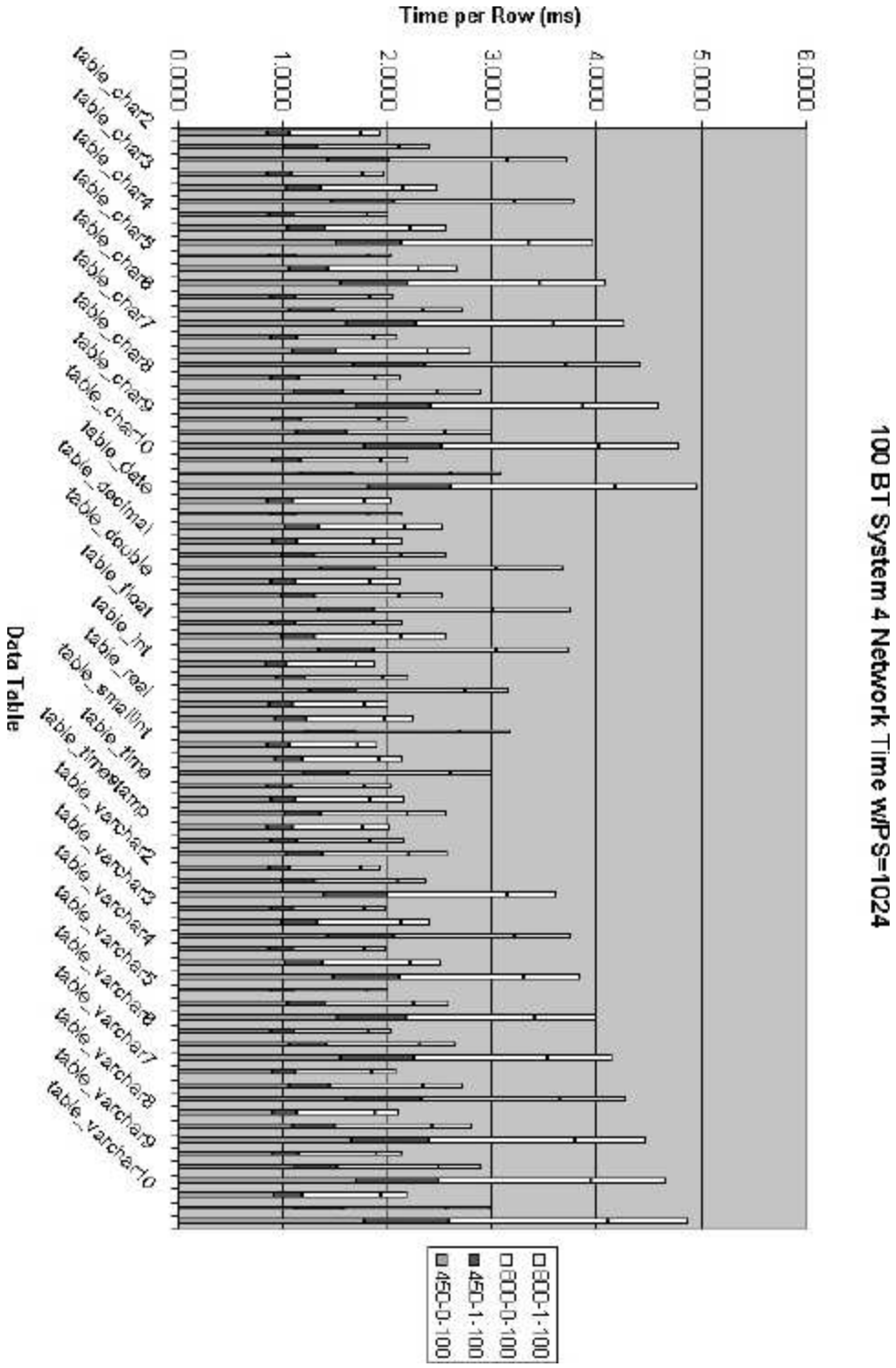

Figure 97: System 4 Network Time with a 100 Base-T link and a packet size of 1024 bytes.