Scalable Performance of System S for Extract-Transform-Load Processing

Toyotaro Suzumura, Toshiaki Yasue, and Tamiya Onodera
IBM Research – Tokyo
1623-14 Shimo-tsuruma, Yamato, Kanagawa, Japan
{toyo, yasue, tonodera}@jp.ibm.com

ABSTRACT

ETL (Extract-Transform-Load) processing is filling an increasingly critical role in analyzing business data and in taking appropriate business actions based on the results. As the volume of business data to be analyzed increases and quick responses are more critical for business success, there are strong demands for scalable high-performance ETL processors. In this paper, we evaluate a distributed data stream processing engine called System S for those purposes. Based on the original motivation of building System S as a data stream processing engine, we first perform a qualitative study to see if the programming model of System S is suitable for representing an ETL workflow. Second we did performance studies with a representative ETL scenario. Through our series of experiments, we found that the SPADE programming model and its runtime environment naturally fits the requirements of handling massive amounts of ETL data in a highly scalable manner.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – Distributed databases; D.3.4 [**Software**]: Processors—*runtime environments*

General Terms

Software Architecture, Scalability, Performance, Experimentation

Keywords

Stream Computing, Database Stream Management System, DSMS, Extract-Transform-Load, System S

1. INTRODUCTION

Demands are growing for ETL (Extract-Transform-Load) processing to analyze business data and support appropriate business responses. In situations where the volume of business data to be analyzed is becoming large and quick response is increasingly critical to effective competition, there are strong needs for scalable, high-performance ETL processing platforms. In this paper, we evaluate a distributed data stream processing engine called System S for such requirements. Based on the original goal to build System S as a data stream processing engine, we first did a qualitative study to see whether System S's programming model is suitable for representing an ETL workflow. Second, we ran a series of performance studies with a representative ETL scenario. Our experiments showed that the SPADE programming model and its runtime environment naturally fit the needs for handling massive amounts of ETL data in a highly scalable manner.

The remainder of the paper is organized as follows. First, the motivation and problem statement of this research are presented in Section 2. Section 3 gives an overview of System S and its suitability for ETL processing. In Sections 4 and 5 we describe the performance study. Discussion appears in Section 6, and related work is reviewed in Section 7, followed by our conclusions and future directions in Section 8.

2. Motivation - High Performance ETL

First we review the key ideas of ETL (Extract-Transform and Load). A data warehouse, a central integrated repository for all of the relevant data, is initially populated using ETL in 3 steps, Extraction, Transformation, and Loading. ETL processes handle the extraction of data from different distributed data sources, cleansing and customizing the data for the business needs and rules, while transforming the data to match the data warehouse schema, and finally loading the data into the data warehouse.

SYSTOR 2010 May 24-26, Haifa, Israel

Data Explosion in ETL

The amount of data stored in a typical contemporary data warehouse may double every 12 to 18 months [31]. IDC [36] analysts believe that unstructured data tends to grow at twice the rate of conventional structured data held in databases. By 2010, this "dark matter", so named due to the challenge of extracting useful information from that form of raw data, will be the majority of all of the enterprise data stored. Most of that dark matter will be in the form of logs for security, network, and system events. Almost everything that happens in a business is recorded in a log file somewhere, making the search and analysis of that data an essential part of managing, securing, and auditing how a company's technology infrastructure is being used. Logs are key to many kinds of regulatory compliance such as PCI, SOX, FISMA, and HIPAA [31].

Examples in various business domains

At the same time, data sources that appear at the initial Extraction stage, are becoming increasingly large, from terabytes to petabytes. For example, it was reported in [32] that the global retail chain Wal-Mart captures point-of-sale transactions from over 2,900 stores in 6 countries and the daily volume of data is more than 7.5 terabytes. Wal-Mart refreshes the sales data hourly and adds a billion rows of data each day, allowing for more complex searches. And the largest—those at or near the 100-terabyte mark—probably triple every three years. Data is also arriving from increasingly complex sources. For example, Wal-Mart has added feeds from RFID (Radio-Frequency Identification) scanners. [33]. Anther Internet auction site, EBay, allows insiders to search auction data for short periods to get deeper insights into customer behaviors [33].

Telecom companies also face challenges in handling massive amount of data, since current smart phones have various sensors and features such as GPS and digital cameras that generate more data than ever before. Also, CDRs (Call Detail Records) are becoming important data sources for analyzing customers' behaviors in traffic analysis, sales campaign management, social network analysis, and so forth [22].

Near-Real Time ETL

Given this data explosion problem, there are strong needs for ETL processing to be as fast as possible so that business analysts can quickly grasp the trends of customer activities.

Wal-Mart [26] refreshes the data in its data warehouse every hour, with 1 billion or more rows of data updated every day. Wal-Mart turned its data warehouse into an operational system for managing daily store operations. Store managers used to query the database at the end of each day to see what was selling at their store. Now they can check hourly and also see what is happening at stores throughout a region, looking for the effects of unusual events such as a snowstorm or hurricane [29]. In addition, business demands such as increased competition, improved sales, better monitoring of customers or goals, precise monitoring of the stock market, and so on can all result in demands for accurate reports and results based on current data and not on yesterday's status [29].

Another crucial problem challenging conventional ways of thinking about ETL is the globalization of the economy, including global commodity markets. The usual nightly ETL processing of the data for morning updates is complicated when an organization's branches are located in different time zones. For these reasons, data warehouses are becoming "active" or "live" data producers for their users [29]. The notion of near real-time ETL has been proposed many times. For example, [29] fully explains the needs for and requirements of near real-time ETL. They review the state of the art for both conventional and near real-time ETL, and discuss the technical issues that arise in the area of near real-time ETL, and they pinpoint interesting research challenges for future work.

Our Motivation

Based on this background and the current needs for ETL processing, here are the motivations of our research:

1. Assess the applicability of System S for data stream processing, considering both qualitative and quantitative ETL constraints.

2. Thoroughly evaluate the performance of System S as a scalable and distributed ETL platform. A consulting company [35] compared the performance of various ETL tools, including open-source and commercial products, but to the best of our knowledge, no research has fully evaluated the performance of an ETL system. In this paper, we focus on the performance characteristics and

scalability of System S for an ETL application. We are not comparing it with other ETL tools since no standard benchmark yet exist, though standardization is an ongoing activity of a TPC committee [18].

3. System S and its suitability for ETL

In this section we first give an overview of System S, a distributed data stream processing engine, and then discuss its suitability for ETL processing from various perspectives such as the programming model, extensibility, and scalability.

3.1 Overview of System S and SPADE

System S [1][2][3][5] is large-scale, distributed data stream processing middleware under development at IBM Research. It handles structured and unstructured data streams and can be scaled to large numbers of compute nodes. The System S runtime can execute a large number of long-running jobs (queries) in the form of data-flow graphs described in its special stream-application language called SPADE (Stream Processing Application Declarative Engine) [1]. SPADE is stream-centric and operator-based for stream processing applications in System S, and also supports all of the basic stream-relational operators with rich windowing semantics. Here are the main built-in operators mentioned in this paper:

- **Functor**: adds attributes, removes attributes, filters tuples, or maps output attributes to a function of input attributes
- **Aggregate**: window-based aggregates, with groupings
- **Join**: window-based binary stream join
- **Sort**: window-based approximate sorting
- **Barrier**: synchronizes multiple streams
- **Split**: splits the incoming tuples for different operators
- **Source**: ingests data from outside sources such as network sockets
- **Sink**: publishes data to outside destinations such as network sockets, databases, or file systems

SPADE also allows users to created customized operations with analytic code or legacy code written in C/C++ or Java. Such an operator is a **UDOP** (User-Defined Operator), and it has a critical role in providing flexibility for System S. Developers can use built-in operators and UDOPs to build data-flow graphs.

After a developer writes a SPADE program, the SPADE compiler creates executables, shell scripts, other configurations, and then assembles the executable files. The compiler optimizes the code with statically available information such as the current status of the CPU utilization or other profiled information. System S also has a runtime optimization system, SODA [5]. For additional details on these techniques, please refer to [1][2][3][5].

SPADE uses code generation to *fuse* operators with PEs. The PE code generator produces code that (1) fetches tuples from the PE input buffers and relays them to the operators within, (2) receives tuples from operators within and inserts them into the PE output buffers, and (3) for all of the intra-PE connections between the operators, it *fuses* the outputs of operators with the downstream inputs using *function calls*. In other words, when going from a SPADE program to the actual deployable distributed program, the logical streams may be implemented as simple function calls (for fused operators) for pointer exchanges (across PEs in the same computational node) to network communications (for PEs sitting on different computational nodes). This code generation approach is extremely powerful because simple recompilation can go from a fully fused application to a fully distributed version, adapting to different ratios of processing to I/O provided by different computational architectures (such as blade centers or Blue Gene).

3.2 Qualitative Evaluation of System S as an ETL platform

In the previous subsection we gave an overview of System S and in this subsection we discuss how the system satisfies recent ETL requirements such as programmability, extensibility, and scalability. .

Programmability

The language for System S, SPADE is especially designed for data stream processing. The essential difference between SPADE and other ETL workflow languages is the notion of "windowing" (a sliding window or tumbling window). System S and other data stream processing systems carefully consider how to create programming models to express the processing for continuously incoming data. The SPADE language uses this windowing mechanism at a language level so that developers can easily handle continuously arriving data. As described in Section

3.1, the *Aggregate* operator is a differencing function that is missing from existing ETL tools. This windowing mechanism is crucially important to realize near-real time ETL processing.

Basically, the other operators that conduct relational algebra are supported by the existing ETL tools that also have rich GUI development environments allowing developers to easily describe their ETL workflows.

Extensibility

As mentioned above, SPADE allows developers to define a customized operator called a UDOP (User-Defined Operator), which is implemented in a procedural language such as C++ or Java. This extensibility enables us to perform more complex analysis such as machine learning algorithms. This is a different from conventional and simple ETL jobs. The existing ETL tools also allow us to define customized code, but not at the same level as SPADE. Source operators can handle multiple heterogeneous data sources including unstructured and structured data. In the current situations where a data warehouse needs to integrate a wider variety of data than before, such as XML data from Web services, highly unstructured data such as voice data from a call center, and raw images from an image sensor.

Applicability for Conventional ETL and Near Real Time ETL

As mentioned in Section 3.1, SPADE has a set of built-in operators that allows us to describe conventional ETL workflows. More importantly, near real-time ETL is close to the original intention of this system, which expects to continuously receive data and process the data on-the-fly in real time or near real time. In general, a stream processing system tends to handle data that comes from sensors via network sockets, but we can extend the general notion to handle flat files or even databases.

Scalability

System S is built with the concept of bringing a scalable and high-performance platform for massive amounts of streaming data. It has a run-time optimization system called SODA [5] as well as compile-time optimizations. Therefore, the scalability is provided naturally by the System S runtime environment.

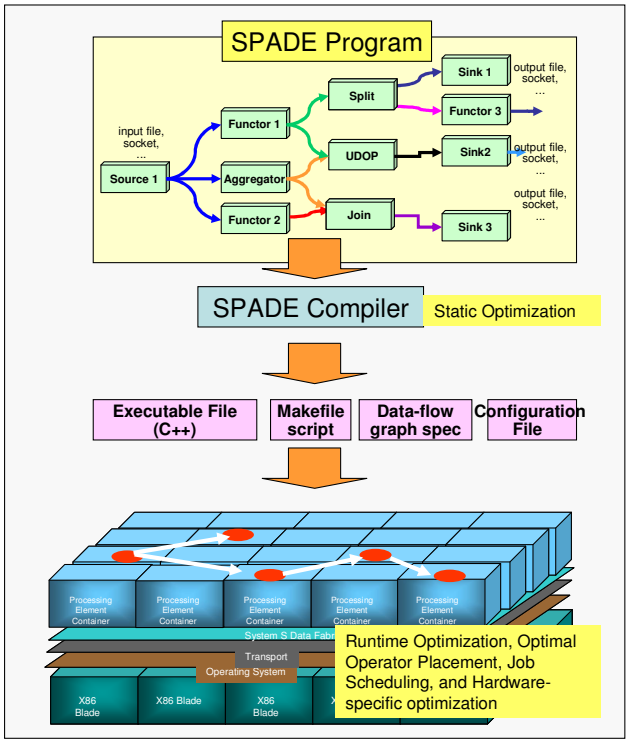


Figure 1. Overview of System S and SPADE

4. Comparing SQL and System S/SPADE with a Moving Average Computation

One of the handy programming models and runtime alternatives to a sophisticated ETL processing platform with rich functionalities is SQL on relational databases. Many enterprise customers are using SQL for simple processing such as sums, averages, maximums, and minimums for the data within certain time windows.

For example, to do such processing with SQL and SPADE, the code would be something like the code in Table 1. This program handles a time-series of stock records whose data schema is comprised of a stock symbol, date, stock closing price, and total share volume (e.g. IBM, 2006-05-14, 8325.11, 3535), and calculates the average, minimum and maximum for closing price, and minimum and maximum for share volume against a sliding window of five days with 1 slide. Since the data contains all of the the share sales records for all of the companies, these computations must be done for each company of interest.

The SPADE program in the table only shows the essential part with the retrieval from the DB using the *ODBCSource* operator, and the actual computation. By using the *Aggregator* operator, this kind of

windowing computation is expressed naturally. The parameters, "pergroup", and "[symbol]" passed to the Aggregator operator specify the computations such as Min, Max, and Avg should be done for each stock symbol.

Another strength of SPADE is the ability to support easy parallelization of such computations, and the distributed version of SPADE is also shown in the table. The *Split* operator is used to split each stock's records based on the stock symbol, so in this case the operator emits 26 different streams by calculating the hash value of the stock symbol. For the split streams, the 26 *Aggregate* operators each receive an incoming stock data stream and perform computations. The node assignment is executed either manually or automatically from the available compute nodes defined in the *[Nodepools]* section. In the manual assignment, if a specify certain operator is specified to run on the 3rd compute node in the node pool, the notation ***->node(pool, 3)*** is added under the operator.

For this simple ETL processing, we compared the performance of SQL and SPADE. The experimental environment we used has 4 nodes, each an Intel Xeon 3.2 GHz core with 3 GB pf RAM, 64-bit hardware, using the RHEL 5.2 OS, Kernel 2.6.180-92. For the relational database, we used IBM DB2 version 9.5.

The results appear in Table 2 as the total elapsed times for processing 500,000 transaction records. For SQL, all of the computations are handled by the database, and only the results are returned. For SPADE, all of the data is retrieved from the database, and the computations are done by SPADE. These two configurations used only 2 of the physical nodes connected via a 1-Gbps network.

	Expression
SQL Version	SELECT **SYMBOL**, **MIN**(CLOSINGPRICE) OVER (PARTITION BY SYMBOL ORDER BY TRADEDATE DESC ROWS 5 PRECEDING), **MAX**(CLOSINGPRICE) OVER (PARTITION BY SYMBOL ORDER BY TRADEDATE DESC ROWS 5 PRECEDING), **AVG**(CLOSINGPRICE) OVER (PARTITION BY SYMBOL ORDER BY TRADEDATE DESC ROWS 5 PRECEDING), **SUM**(CLOSINGPRICE) OVER (PARTITION BY SYMBOL ORDER BY TRADEDATE DESC ROWS 5 PRECEDING), **MIN**(VOLUME) OVER (PARTITION BY SYMBOL ORDER BY TRADEDATE DESC ROWS 5 PRECEDING), **MAX**(VOLUME) OVER (PARTITION BY SYMBOL ORDER BY TRADEDATE DESC ROWS 5 PRECEDING) FROM STOCKREPORT
SPADE Version	stream SourceDataStream(schemaFor(MySchema)) := **ODBCSource**()[connection:"DB2Person"; access:"StockReport"; initDelay:3]{} stream MovingAverage (schemaFor(aggregatedData)) :=**Aggregate**(SourceDataStream <count(5),count(1),**pergroup**>)[**symbol**] {Any(symbol), Cnt(), MCnt(), Min(closingprice), Max(closingprice), Avg(closingprice), Min(volume), Max(volume)}
Distri-buted SPADE Version	[Nodepools] nodepool sp[1] := ("s72x336-03") nodepool np[4] := ("s72x336-00", "s72x336-00", "s72x336-02", "s72x336-04") for_begin @index 1 to 26 stream SplitDataStream@index (schemaFor(MySchema)) := **Split**(SourceDataStream) [hashCode(toCharacterCode(symbol, 0) - toCharacterCode("A", 0))] {} -> node(sp, 0), stream MovingAverage@index (schemaFor(aggregatedData)) := **Aggregate**(SplitDataStream@index <count(5), count(1)>)[] {Any(symbol),Cnt(),MCnt(), Min(closingprice), Max(closingprice), Avg(closingprice), Min(volume), Max(volume)} -> node(np, +) for_end

Table 1 Expression for SQL and SPADE

The distributed version of SPADE is computed on 3 nodes (of the 4 cores), and its results are improved from 18.85 to 10.75 seconds. When compared to the performance of SQL, the distributed version of SPADE is not as good as we expected even though it ran on multiple nodes. This is because the computations in this scenario are relatively light compared to the network overhead of sending the data to the different physical nodes. However, these results do show that SPADE/System S can easily be used in a simple ETL processing scenario with good performance.

	SQL	SPADE (1 node)	Distributed Version of SPADE
Elapsed Time	13.965 (s)	18.85 (s)	10.75 (s)
Throughput (records/sec)	35,803	26,525	46,511

Table 2. Performance Comparison between SQL and SPADE

5. Performance Evaluation of System S as a Distributed ETL Platform

In this section we report on a performance evaluation for a representative ETL scenario using System S. This is our Experiment II.

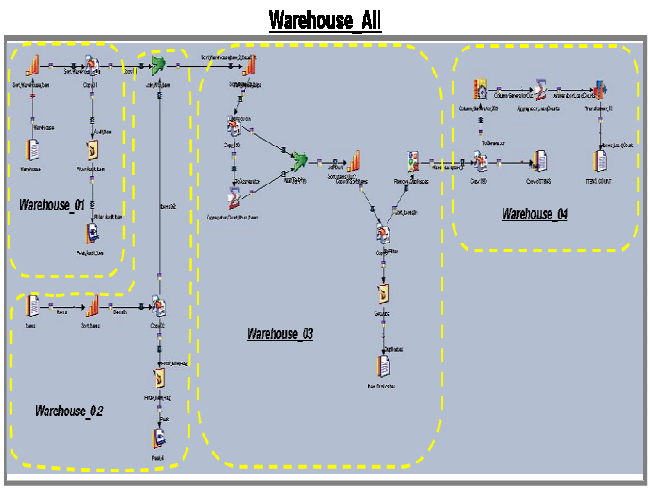


Figure 2. Target ETL Scenario

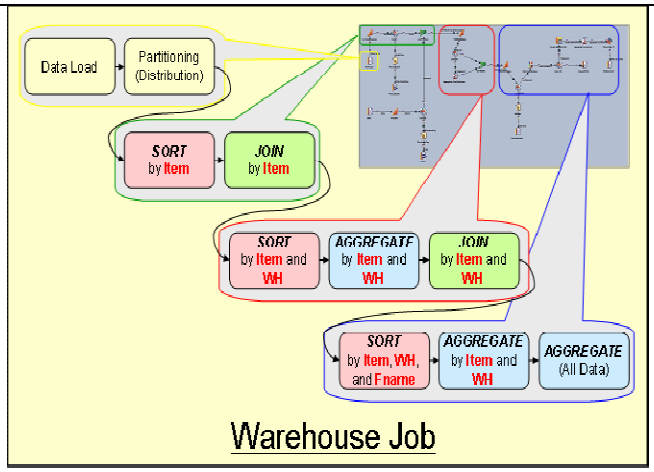


Figure 3. Core Processing in the target ETL scenario

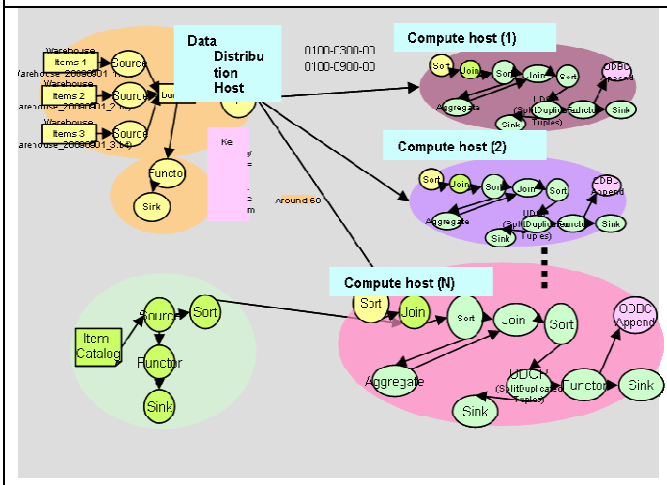


Figure 4. SPADE Program for Experiment I

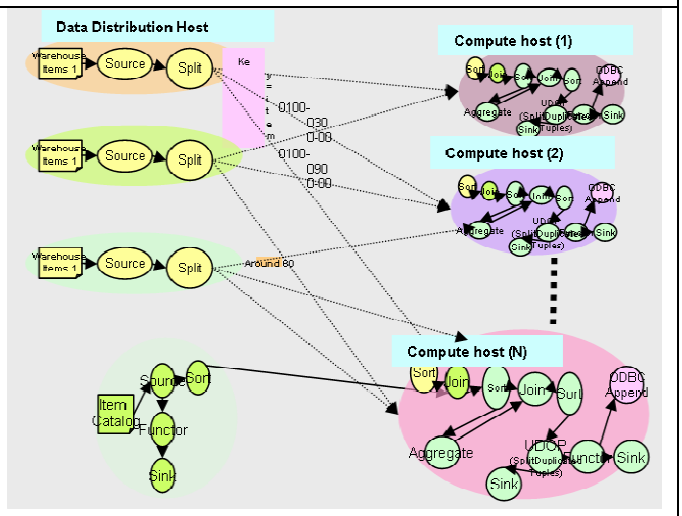


Figure 5. SPADE Program for Experiment II

5.1 Target ETL Scenario

Since there is no standard benchmark for comparing ETL middleware, for our target ETL scenario, we selected an ETL scenario named *Warehouse*, which does inventory processing for multiple warehouses. This scenario was selected as representative based on discussions with practitioners in our company who have worked on ETL-related customer engagements. This application uses our DataStage [37] product, which handles most of the required ETL processing.

This ETL scenario consists of four jobs, each of which handles certain stages of the analysis. Here is the processing for each job:

- **Warehouse_01**: Extract millions of data records for each warehouse, sort the data by item id, and store the result in a data-set file. Each data record in this output file has 8 elements (with the SQL type in parentheses): item (VarChar), warehouse id (Integer), stock (Integer), and 5 other elements (which are not relevant for this report). The main processing at this stage is the sorting of all of the items by item name. The inventory data was stored in multiple files.
- **Warehouse_02**: Join the item data read from the output file written by Warehouse_01, with item description data stored in an item description file that contains pairs of an item name and the description of the item (approximately 3,000 item descriptions), and output another file. The main processing at this stage is the join operation.
- **Warehouse_03**: Read the data-set file written by Warehouse_02, count up the number of the items stored in each warehouse, and produces a unique pair of item data and warehouse by remove the

duplications. Both the unique data and duplicated data are stored into two seperate file.

- **Warehouse_04**: Read the data-set file written by Warehouse_03, and load the data and the total number of Item into DB.

We made an integrated version called **Warehouse_All** illustrated in Figure 2, consisting of these four jobs for the performance evaluation to eliminate the file I/O overhead between the jobs. Figure 3 illustrates the essential ETL operations in this scenario.

5.2 Experimental Environment and Data Set

We used a set of 14 compute nodes, each of which is a 3.0-GHz Intel Xeon X5365 with 4 physical cores with Hyper-Threading, and 16 GB of RAM running RedHat Enterprise Linux 5.3 64 bit (kernel 2.6.18-164.el5). The hostname of each node (e.g. e0101b05e1) starts with "e0101b" followed by a variable number from 01 to 14, and the constant string "e1". All of the nodes are connected via either a 1-GB Ethernet LAN or Infiniband Network (DDR 20Gbps). In the case of 1Gb Ethernet, the network latency was 0.047 ms, and the actual network throughput measured with *netperf* was 941 Mbits/second. The data set we used for the experiments is 9 million records that accounts for 1 around GByte data since individual item record is around 100 byte.

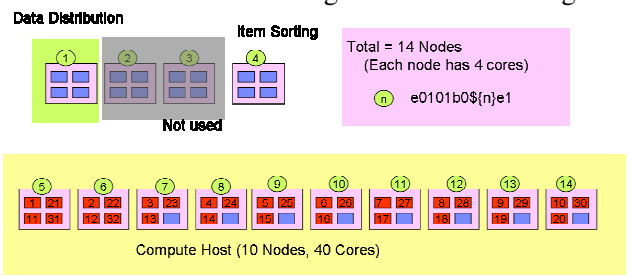
5.3 Developing Target ETL Scenario by SPADE

Developing the ETL scenario described in the previous section by SPADE is straightforward. The overall data flow diagram that represents the SPADE program is illustrated in Figure 4, and the SPADE code is described in the Appendix section. Although SPADE allows developers to define their user-defined operator written in C++ or Java, the resultant SPADE program leverages built-in operators such as Source, Split, Functor, and so forth. The only UDOP (User-defined Operator) operator (named "FilterDuplicatedItems" in the SPADE program) we need to develop in C++ is the one that omits the duplicated items and emits the only set of unique items in the Warehouse_02 stage.

Parallelization is also done by using the Split operator and value range partitioning since the identifier field of each warehouse item is within the certain rage. In front of the Split operator, there is a Functor operator that computes the appropriate node number from the identifier of an item. The Split operator splits out all the items to multiple operators, each of which is executed on different compute nodes.

5.4 Experiment I

For the first experiment of the above ETL scenario, we conduct the performance evaluation without any optimization. The experimental environment is comprised of 1 source node for data distribution, 1 node for item sorting in the Warehouse_02 stage, and 10 nodes for computation that processes all the operations from Warehouse_01 to Warehouse_04 except for item sorting mentioned above. Each compute node has 4 cores and we manually allocate each operator with the scheduling policy described in the following diagram.



This diagram shows the case in that 32 operators are used for the computation in compute hosts. Each operator is allocated to adjunct node and processing core in order rather than filling up all the processing core in each node and moving to the next node. The only 1 node (Node 1) is responsible for distributing data to compute nodes.

Analysis (I-a) Throughput and Breakdown the total time

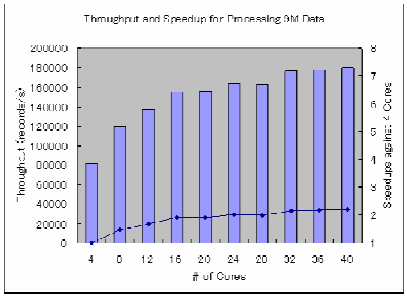


Figure 6. Analysis I-a for Experiment I

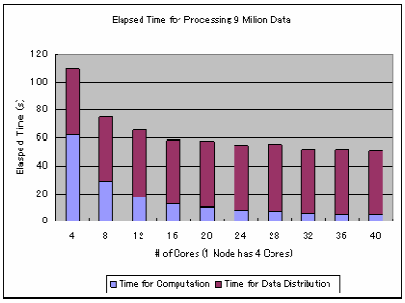


Figure 7. Analyais I-a for Experiment I.

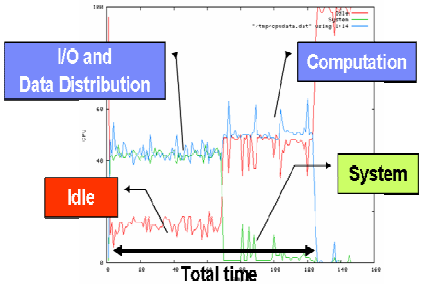


Figure 8. Profiling in compute node in Analysis I-a

The Figure 6 shows the throughput data (records per second) with varying number of compute cores. The scalability can be observed up to 16 cores, the throughput is saturated around 20 cores. Figur 7 includes the breakdown of the time for computation and data distribution. Obviously, the data distribution is becoming dominant with many cores. Figure 8 is the profiling result obtained in certain compute node. This result validates that I/O and data distribution can be occurred in the half of the total time by looking at the CPU utilization of kernel, and the computation has not been started until all the required data set has arrived.

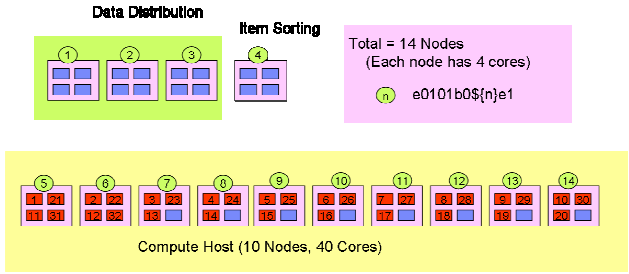
Analysis (I-b) Speed-up ratio against 4 cores when focusing on only "computation part"

In Figure 9, when you focus on the throughput of only computation part, the resulting data shows more than linear scalability since the given number of data to be passed to the Sort operator in the first stage (Warehouse_01) is decreased by parallelization and partitioning, and the computational complexity of Sorting is around *O(nlogn)*.

5.5 Experiment II

In the previous experiment (Experiment I) , we find out that most of the time is spent in the data distribution and I/O processing. In this experiment (II), multiple nodes are participated in the data distribution while each source operator is only responsible for certain chunk of data records divided by the number of source operators.

As an experimental environment for this experiment, 3 nodes, 01, 02, 03, are dedicated for I/O processing and each node distributes 3 million data records, and , 10 nodes from 05 to 14 (each node has 4 physical cores) are used for computation**.** The scheduling policy is shown in the following diagram in the case that 32 compute operators are allocated.



The SPADE program is changed to the data flow diagram shown in Figure 5. In this experiment, 3 nodes are participated in the data distribution and the varying number of compute cores/nodes from 4 to 40 cores. Since 3 nodes are participated in the data distribution, the number of communication is increased to 3 x N (# of compute nodes) from N in the previous experiment. We used the Infiniband network since this topology requires three times more communication and the inter-connect communication is highly important.

The result is shown in Figure 10. When 3 nodes are participated in the data distribution, the throughput is increased to almost double when compared with the result given by Experiment I. This is because the computation can start earlier than the previous experiment, and the data distribution cost becomes relatively lower in the total time.

Analysis (II-a) Optimization by changing the number of source operators

Although the throughput becomes much better than Experiment I, the throughput is saturated around 16 cores in Figure 10 due to the lack of data feeding ratio against computation. We changed the number of source operators (3, 5, 9, 15) while not changing the total volume of data (9 million data records), and measured throughput. This result is shown in Figure 11. In this experiment, we only tested 32 operators for computation and the result shows that the one with 9 source operators for data distribution obtains the best

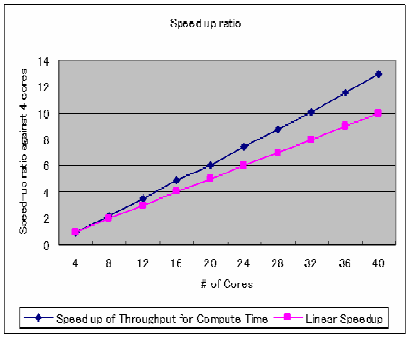


Figure 9. Analysys I-b in Experiment I

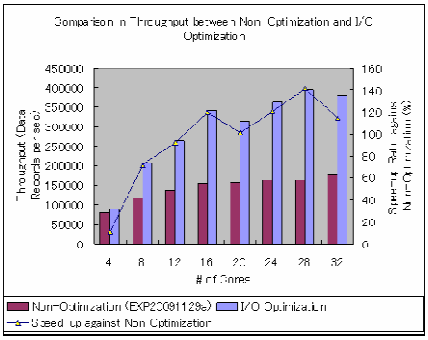


Figure 10. Experiment II: Comparison between Experiment I and Experiment II

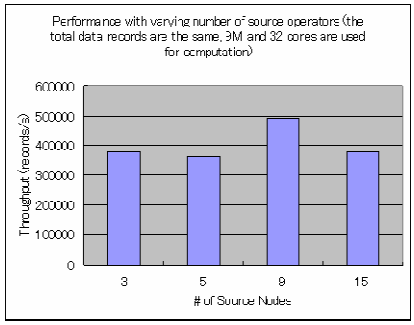


Figure 11. Analysis II-a in Experiment II: Different number of source nodes

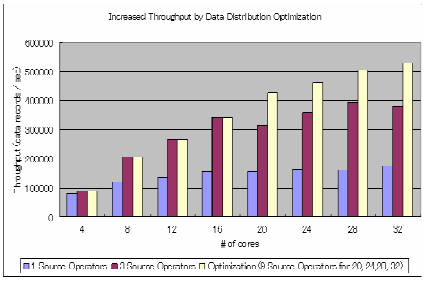


Figure 12. Analysis II-b in Experiment II: Optimized Throughput by Data Distribution

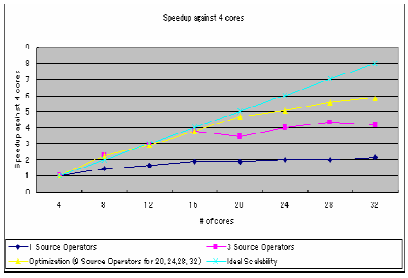


Figure 13. Analysis II-b in Experiment II

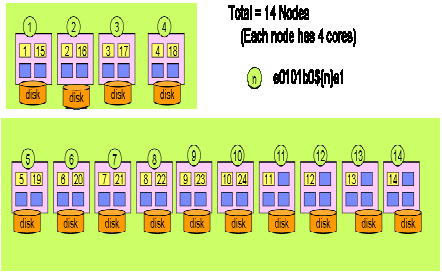
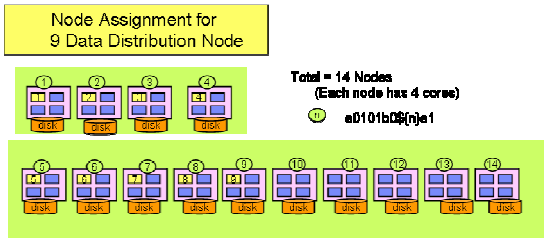


Figure 14. Node Assignment for Experiment III

throughput. 9 source operators are allocated in such a way in the following diagram.



Analysis (II-b) : Increased Throughput by Data Distribution Optimization

The graph shown in Figure 12 demonstrates the overall results by taking the same optimization approach in previous experiment, which increases the number of source operators. 3 source operators are used in the case of 4, 8, 12, 16 compute operators, and 9 source operators are used for 20, 24, 28 and 32 compute operators. We achieved **5.84 times** speedup against 4 cores at 32 cores. In Figure 13, the line labeled "optimization" shows the best performance since 9 nodes are participated in the data distribution for 20, 24, 28 and 32 computational cores.

5.6 Experiment III – Data Distribution Optimization

In this experiment, we test the performance speed-ups by increasing more source operators than previous experiment (Experiment II). We also identify the performance comparison between Infiniband network and the commodity 1Gbps network although previous experiments use the Infiniband network.

Firstly we increase the number of source operators up to 45 from 3, and test this configuration against relatively large number of compute cores, 20, 24, 28, and 32 cores. The node/core assignment for data distribution and computation is the same as previous experiment (Experiment II). For instance, all the 14 nodes participate in the data distribution, and each Source operator is assigned as the manner described in Figure 14. When 24 *Source* operators are allocated to each node in order, when 14 source operators are already allocated to 14 nodes, then the next source operator is allocated to the first node 1.

Each *Source* operator reads the number of records that divide the total data records (9M recordss) with the number of *Source* operators. This data division is conducted in prior to the actual runtime using a Linux tool called "split".

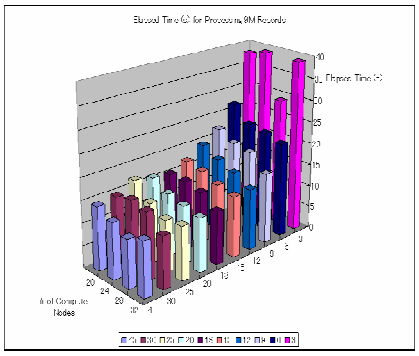


Figure 15. Experiment III: Elapsed Time with varying number of compute nodes and source operators

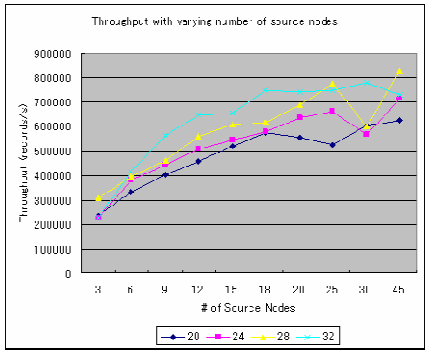


Figure 16. Experiment III: Throughput W/ Infiniband

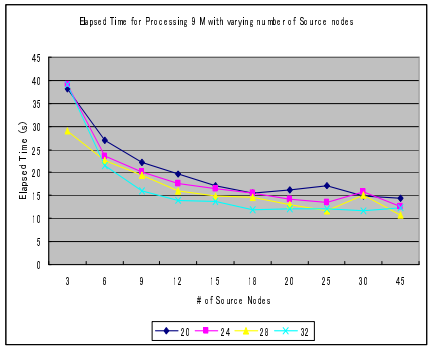


Figure 17. Experiment III: Elapsed Time W/ Infiniband

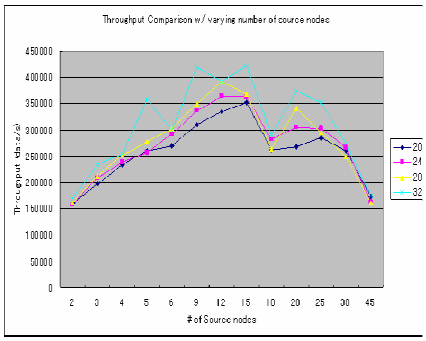


Figure 18. Analysis III-a in Experiment III: Throughput W/O Infiniband

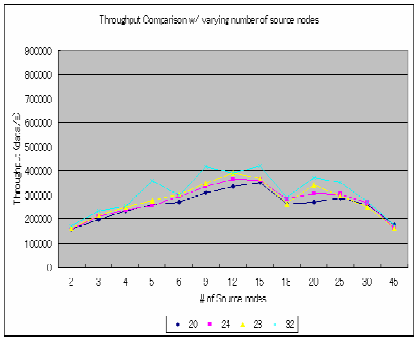


Figure 19. Analysis III-a in Experiment III: Elapsed Time W/O Infiniband

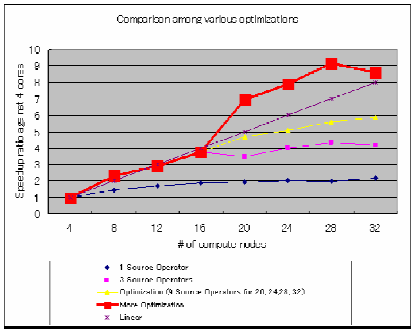


Figure 20. Analysis III-b in Experiment III: Optimized Performance Result

Overall Result for Experiment III

The 3D graph shown in Figure 15 shows the total elapsed time against the combination of certain number of compute cores (we chose relatively large compute cores, 20, 24, 28, and 32) and source operators from 3 to 45. The Z-axis is the elapsed time and the graph clearly shows that the elapsed time is decreased by increasing the number of source operators. Figure 16 and Figure 17 shows the throughput and elapsed time respectively with varying number of source operaters from 3 to 45 in X-axis. The throughput is saturated at a certain point around 18 source operators.

Analysis (III-a) Performance without Infiniband

We also measured the throughput without the Infiniband network against varying number of source operators. The throughput and elapsed time with varying number source operators from 3 to 45 are shown in Figure 18 and Figure 19. Unlike the performance we obtained with Infiniband, the throughput is saturated between 12 and 15 source operators.

This result shows that the throughput is around 400000 data records per seconds at maximum, and this accounts for around 360 Mbps. Although the network we used in this experiment is 1Gbps, this assumes to be an upper limit for consuming full network bandwidth and the middleware overhead of System S. The performance degradation can be observed between 15 and 18 source operators, and we assume that this is because, 14 source operators are allocated to 14 nodes and afterwards 2 or more operators (processes) simultaneously accesses the 1Gbs network card and the resource contention is occurred. This differs from the behavior seen in the result with Infiniband. Thus, when comparing the throughput by enabling the Infiniband network, the absolute throughput number when enabling Infiniband is almost "**double**" against w/o Infiniband. This result indicates that using Infiniband in ETL-typed workloads is essential to obtain high throughput.

Analysis (III-b) Comparison between optimized and other versions

The bar in bold in Figure 20 shows the speed-ups ratio against 4 cores when appropriate number of

source operators are used and data distribution is optimized. With the comparison to linear speedups (labeld in "Linear" in the graph), the optimized version demonstrated nearly super-linear speed-ups. Theoretically, when an ETL scenario includes the Sorting operation that needs *O(nlogn)* computational complexity, the speed-up becomes super-linear. Although the optimized version shows great performance over other experimental results, it has potentiality to obtain more speed-ups when considering theoretical potential speed-ups. Especially when the number of source operators is 45, the performance is greatly saturated. Moreover, when we tested 90 source operators, we could not even get the result. For this matter, we need further investigation.

6. Discussion - Towards Data Distribution Optimization

A series of experimental results in the previous section have shown that data distribution cost is dominant in such an ETL scenario that requires the barrier synchronization that needs all the data to be prepared for further processing such as the *Sort* operation, and the final result after a series of data distribution optimization shows the significant speed-up. This result indicates that it is critically important to minimize the cost for data distribution.

Meanwhile, our study shows that the usage of the Infiniband network dramatically accelerates the data distribution when compared to the commodity 1Gbps network. We also find out that when changing the number of data feed (or source) operators, the throughput is dramatically increased, and we eventually obtained super-linear scalability shown in Figure 20. However, the number of source operators to achieve maximum throughput depends on the available network facility (Infiniband, 1Gbps/10Gbps Ethernet, etc) since the experimental results with the 1Gbps and Infiniband shows different performance characteristics.

We identified the appropriate number of source operators through a series of long-running experiments. However, it would not be reasonable for such a distributed system as System S to force users/developers to experimentally find the appropriate number of source nodes. We will need to have an automatic optimization mechanism that maximizes the throughput by automatically finding the best number of source nodes in a seamless manner from the user. The current SPADE compiler has compile-time optimizer by obtaining the statistical data such as tuple/byte rates and CPU ratio for each operator and allocates each operator to appropriate compute nodes.

For future work, we would like to let users/developers to write a SPADE program without considering the data partitioning and data distribution. By extending the current System S optimizer [5], the system automatically could convert the user's original SPADE program to achieve the maximum data distribution.

7. Related Work

Near Real-Time ETL[21][22][23][24][25][26]: Data warehouses are traditionally refreshed in a periodic manner, most often on a daily basis. Thus, there is some delay between a business transaction and its appearance in the data warehouse. The most recent data is trapped in the operational sources where it is unavailable for analysis. For timely decision making, today's business users ask for ever fresher data. Near real-time data warehousing addresses this challenge by shortening the data warehouse refreshment intervals and hence, delivering source data to the data warehouse with lower latency. One consequence is that data warehouse refreshment can no longer be performed in off-peak hours only. Panons. et.al[21] reviewed the state of the art of both conventional and near real time ETL, and they discuss the background, the architecture, and the technical issues that arise in the area of near real time ETL, and they pinpointed interesting research challenges for future work.

Tomas. et.al [23] points out that the source data may be changed concurrently to data warehouse refreshment. They show that anomalies may arise under these circumstances leading to an inconsistent state of the data warehouse and they propose approaches to avoid refreshment anomalies

ETL Benchmarking [18][19]: Wyatt. et.al [1] identify a common characteristics of ETL workflows in an effort of proposing a unified evaluation method for ETL, and identifieed the main points of interest in designing, implementing, and maintaining ETL workflows. They also propose a principled organization of test suites based on the TPC-H schema for the problem of experimenting with ETL workflows.

8. Conclusions and Future Directions

In this paper we evaluate System S as a scalable ETL platform from both a qualitative and quantitative perspective. To implement our representative ETL scenario with System S and its programming language called SPADE, we identified that SPADE has sufficient expressiveness for describing ETL workflows. For its performance and scalability, we conducted a series of experiments, but the key optimization is found to be data distribution part. By optimizing the number of data feed nodes, the most optimized throughput shows super-linear scalability.

For future work, we will extend SPADE/System S to have automatic optimization for data distribution described in the discussion section. We will also extend this performance study using a benchmark which will appear in the near future like the one under discussion.

Acknowledgements

We would like to acknowledge Nagui Halim, Mitch Cohen, and other team researchers at the IBM Watson Research Center, the principal investigators of the System S project for providing us with invaluable guidance.

REFERENCES

[1] Bugra Gedik, Henrique Andrade, et al., SPADE: the system s declarative stream processing engine, Proceedings of the 2008 ACM SIGMOD internal conference on Management of data.

[2] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: A distributed, scalable platform for data mining", in Workshop on Data Mining Standards, Services and Platforms, DM-SSP, Philadelphia, PA, 2006.

[3] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani, "Design, implementation, and evaluation of the linear road benchmark on the stream processing core", in International Conference on Management of Data, ACM SIGMOD, Chicago, IL, 2006.

[4] K.-L. Wu, P. S. Yu, B. Gedik, K.W. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. A. George, X. Gu, G. Luo, and H. Wang, "Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S", in VLDB, 2007.

[5] J. L. Wolf, et.al, "SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems," Proc. Of the ACM/IFIP/USENIX 9th Int. Middleware Conference, Middleware 2008, Leuven, Belgium, Dec. 2008.

[6] Chandrasekaran et al.: TelegraphCQ:Continuous Dataflow Processing for an Uncertain World, CIDR 2003

[7] Cranor et al: Gigascope: High Performance Network Monitoring with an SQL Interface, SIGMOD 2002

[8] Daniel J. Abadi, et.al, The Design of the Borealis Stream Processing Engine, 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05), Asilomar, CA, January 2005

[9] Arasu, A. and Babcock, B. and Babu, S. and Cieslewicz, J. and Datar, M. and Ito, K. and Motwani, R and Srivastava, U. and Widom, J., STREAM: The Stanford Data Stream Management System (2004), Technical Report.

[10] Bouillet, E. et.al ,Data Stream Processing Infrastructure for Intelligent Transport Systems Vehicular Technology Conference, 2007. VTC-2007 Fall. 2007 IEEE 66th Volume, Issue, Sept. 30 2007-Oct. 3 2007 Page(s):1421 – 1425

[11] Baby & Widom: An Adaptive Engine for Stream Query Processing, SIGMOD 2004

[12] Bugra Gedik, Kun-Lung Wu, and Philip S. Yu. Efficient Construction of Compact Source Filters for Adaptive Load Shedding in Data Stream Processing. In Proceedings of the 24th IEEE International Conference on Data Engineering, (IEEE ICDE), April 7-12, 2008.

[13] Arvind Arasu et al., The CQL Continuous Query Language: Semantic Foundations and Query Execution, VLDB Journal 2003.

[14] Ahmad et al.: StreaMon: An Adaptive Engine for Stream Query Processing, SIGMOD 2005 (distributed streams).

[15] Carney et al.: Monitoring Streams – A New Class of Data Management Applications, VLDB 2003.

[16] A. Simitsis, P. Vassiliadis, and T. Sellis. Optimizing ETL processes in data warehouses. In ICDE'05: Proceedings of the 21st International Conference on Data Engineering, pages 564--575, Washington, DC, USA, 2005. IEEE Computer Society.

[17] V. Tziovara, P. Vassiliadis, and A. Simitsis. Deciding the physical implementation of etl workflows. In DOLAP '07: Proceedings of the ACM tenth international workshop on Data warehousing and OLAP, pages 49--56, New York, NY, USA, 2007. ACM.

[18] Principles for an ETL Benchmark, Wyatt, Len; Caufield, Brian; Pol, Daniel: Performance Evaluation and Benchmarking, Lecture Notes in Computer Science, Volume 5895. ISBN 978-3-642-10423-7. Springer-Verlag Berlin Heidelberg, 2009, p. 183,

[19] Benchmarking ETL Workflows, Simitsis, Alkis; Vassiliadis, Panos; Dayal, Umeshwar;Karagiannis, Anastasios; Tziovara, Vasiliki,

[20] P. Vassiliadis, A. Karagiannis, V. Tziovara, and A. Simitsis. Towards a benchmark for etl workflows,. In QDB, pages 49--60, 2007.

[21] Panons Vassiliadis, Alkis Simitsis, Near Real Time ETL,

[22] Munir Cochinwala, etc. Near Real-time Call Detail Record ETL Flows, VLDB2009, BIRTE Workshop, BIRTE 2009

[23] Thomas Jorg and Stefan Dessloch, Nea Real-Time Data Warehousing Using State-of-the-Art ETL Tools, BIRTE2009 Workshop in VLDB2009

[24] M. Asif Naeem, et.al. Comparing Global Optimization and Default Settings of Stream-based Joins, BIRTE2009 Workshop in VLDB2009

[25] Maik Thiele, et.al, Evaluation of Load Scheduling Strategies for Real-Time Data Warehouse Environment, BIRTE2009 Workshop in VLDB22009,

[26] Irina Botan, et.al, Federated Stream Processing Support for Real-Time Business Intelligence Applications, BIRTE2009 Workshop in VLDB 2009

[27] http://www.informationweek.com/news/global-cio/showArticle.jhtml?articleID=175801775

[28] Eric KNORR, Dealing with the data explosion, http://www.infoworld.com/d/storage/dealing-data-explosion-690, 2009/8

[29] Pasnon Vassiliadis and Alkis Simitsis, Near Real Time ETL

[30] Thomas Jorg and Stefan Dessloch, Near Real-Time Data Warehousing Using State-of-the-Art ETL Tools

[31] http://www.infoworld.com/d/storage/dealing-data-explosion-690

[32] http://www.informationweek.com/news/global-cio/showArticle.jhtml?articleID=175801775

[33] http://www.informationweek.com/news/global-cio/showArticle.jhtml?articleID=175801775

[34] http://www.informationweek.com/news/global-cio/showArticle.jhtml?articleID=175801775

[35] MANAPPS ETL Benchmark (2008/10) http://marcrussel.files.wordpress.com/2008/10/etlbenchmarks_manappsc221008.pdf

[36] IDC: http://www.idc.com/

[37] DataStage:http://www-06.ibm.com/software/jp/data/infosphere/datastage/

Appendix: SPADE Program

```
[Nodepools]
nodepool np[] := ("e0101b01e1","e0101b02e1","e0101b03e1","e0101b04e1", .. "e0101b014e1")
[Program]
vstream Warehouse1Schema(id: Integer, item : String, Onhand : String, allocated : String, hardAllocated : String, fileNameColumn : String)
vstream Warehouse2OutputSchema(id: Integer, item : String, Onhand : String, allocated : String, hardAllocated : String, fileNameColumn : String, description: StringList)
vstream ItemSchema(item: String, description: StringList)

################ warehouse 1 ######################
bundle warehouse1Bundle := ()
for_begin @i 1 to 3
stream Warehouse1Stream@i(schemaFor(Warehouse1Schema))
 := Source()["file:///SOURCEFILE", nodelays, csvformat]{}
      -> node(np, 0), partition["Sources"]
warehouse1Bundle += Warehouse1Stream@i
for_end

## stream for computing subindex
stream StreamWithSubindex(schemaFor(Warehouse1Schema), subIndex: Integer)
      := Functor(warehouse1Bundle[:])[] {
 subIndex := (toInteger(strSubstring(item, 6,2)) / (60 / COMPUTE_NODE_NUM))-2 }

for_begin @i 1 to COMPUTE_NODE_NUM
  stream ItemStream@i(schemaFor(Warehouse1Schema), subIndex:Integer)
for_end
  := Split(StreamWithSubindex) [ subIndex ]{}

for_begin @i 1 to COMPUTE_NODE_NUM
 stream Warehouse1Sort@i(schemaFor(Warehouse1Schema))
  := Sort(ItemStream@i <count(SOURCE_COUNT@i)>)[item, asc]{}
stream Warehouse1Filter@i(schemaFor(Warehouse1Schema))
  := Functor(Warehouse1Sort@i)[ Onhand="0001.000000" ] {}
Nil := Sink(Warehouse1Filter@i)["file:///WAREHOUSE1_OUTPUTFILE@i",
                csvFormat, noDelays]{}
for_end

 ################ warehouse 2 ######################
stream ItemsSource(schemaFor(ItemSchema))
  := Source()["file:///ITEMS_FILE", nodelays, csvformat]{}
stream SortedItems(schemaFor(ItemSchema))
  := Sort(ItemsSource <count(ITEM_COUNT)>)[item, asc]{}
for_begin @i 1 to COMPUTE_NODE_NUM
 stream JoinedItem@i(schemaFor(Warehouse2OutputSchema))
  := Join(Warehouse1Sort@i <count(SOURCE_COUNT@i)>;
                SortedItems <count(ITEM_COUNT)>)
      [ LeftOuterJoin, {item} = {item} ]{}

################ warehouse 3 and 4 ######################
for_begin @i 1 to COMPUTE_NODE_NUM
 stream SortedItems@i(schemaFor(Warehouse2OutputSchema))
  := Sort(JoinedItem@i <count(JOIN_COUNT@i)>)[id, asc]{}

stream AggregatedItems@i(schemaFor(Warehouse2OutputSchema), count: Integer)
  := Aggregate(SortedItems@i <count(JOIN_COUNT@i)>)
    [item . id]
    { Any(id), Any(item), Any(Onhand), Any(allocated),
      Any(hardAllocated), Any(fileNameColumn), Any(description), Cnt() }

 stream JoinedItem2@i(schemaFor(Warehouse2OutputSchema), count: Integer)
  := Join(SortedItems@i <count(JOIN_COUNT@i)>;
       AggregatedItems@i <count(AGGREGATED_ITEM@i)>)
     [ LeftOuterJoin, {id, item} = {id, item} ] {}

 stream SortJoinedItem@i(schemaFor(Warehouse2OutputSchema), count: Integer)
  := Sort(JoinedItem2@i <count(JOIN_COUNT@i)>)[id(asc).fileNameColumn(asc)]{}

 stream DuplicatedItems@i(schemaFor(Warehouse2OutputSchema), count: Integer)
 stream UniqueItems@i(schemaFor(Warehouse2OutputSchema), count: Integer)
 := Udop(SortJoinedItem@i)["FilterDuplicatedItems"]{}
 stream FilterStream@i(item: String, recorded_indicator: Integer)
 := Functor(UniqueItems@i)[] { item, 1 }
stream AggregatedItems2@i(LoadNum: Integer, Item_Load_Count: Integer)
 := Aggregate(FilterStream@i <count(UNIQUE_ITEM@i)>)
    [ recorded_indicator ] { Any(recorded_indicator), Cnt() }
 stream AddTimeStamp@i(LoadNum: Integer, Item_Load_Count: Integer, LoadTimeStamp: Long)
 := Functor(AggregatedItems2@i)[] { LoadNum, Item_Load_Count, timeStampMicroseconds() }
 Nil := Sink(AddTimeStamp@i)["file:///final_result.out", csvFormat, noDelays]{}
 for_end
```