2016

Efficient Partitioning and Allocation of Data for Workflow Compositions

Annamaria Victoria Kish
University of South Carolina

Efficient Partitioning and Allocation of Data for Workflow Compositions

by

Annamaria Victoria Kish

Bachelor of Arts
University of Connecticut 1980
Master of Science
University of South Carolina 2006

Submitted in Partial Fulfillment of the Requirements

for the Degree of Doctor of Philosophy in

Computer Science and Engineering

College of Engineering and Computing

University of South Carolina

2016

Accepted by:

Csilla Farkas, Major Professor

Caroline Eastman, Committee Member

Manton Matthews, Committee Member

John Rose, Committee Member

Eva Czabarka, Committee Member

Lacy Ford, Senior Vice Provost and Dean of Graduate Studies

Abstract

Our aim is to provide efficient partitioning and allocation of data for web service compositions. Web service compositions are represented as partial order database transactions. We accommodate a variety of transaction types, such as read-only and write-oriented transactions, to support workloads in cloud environments. We introduce an approach that partitions and allocates small units of data, called micropartitions, to multiple database nodes. Each database node stores only the data needed to support a specific workload. Transactions are routed directly to the appropriate data nodes. Our approach guarantees serializability and efficient execution.

In Phase 1, we cluster transactions based on data requirements. We associate each cluster with an abstract query definition. An abstract query represents the minimal data requirement that would satisfy all the queries that belong to a given cluster. A micropartition is generated by executing the abstract query on the original database.

We show that our abstract query definition is complete and minimal. Intuitively, completeness means that all queries of the corresponding cluster can be correctly answered using the micropartition generated from the abstract query. The minimality property means that no smaller partition of the data can satisfy all of the queries in the cluster.

We also aim to support efficient web services execution. Our approach reduces the number of data accesses to distributed data. We also aim to limit the number of replica updates. Our empirical results show that the partitioning approach improves data access efficiency over standard partitioning of data.

In Phase 2, we investigate the performance improvement via parallel execution.

Based on the data allocation achieved in Phase I, we develop a scheduling approach. Our approach guarantees serializability while efficiently exploiting parallel execution of web services.

We achieve conflict serializability by scheduling conflicting operations in a predefined order. This order is based on the calculation of a minimal delay requirement. We use this delay to schedule services to preserve serializability without the traditional locking mechanisms.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation is concerned with correct partitioning and scheduling for a web service execution in a parallel database environment. The core idea is to design a system in the spirit of just-in-time inventory control where only the data required for current processes are available; it is a fragmentation technique based upon the concept of minimal data sets. We are also interested in developing a scheduling method that that provides efficient concurrency control for web services executing in a parallel environment.

1.1 Problem Statement

There is a need to provide efficient data partitioning and efficient scheduling for transactional web services and web service compositions in the context of parallel systems. Current web services solutions for partitioning are based on full replication [69] [13]. However this approach involves global concurrency control and replica updates, which can be costly [50]. Current web services scheduling techniques focus on quality of service properties [70] and on controlling communications costs [7]. However transactional properties of composite web services have not been fully explored with respect to effective scheduling of web services.

Full replication is used as a strategy to improve availability of transactional data. The expense of providing full replication grows when more services, more users, and more databases are on a system. We are motivated to provide a solution that provides

good data availability on systems that scale. Most services access just a portion of the data from a given database, as in Figure 1.1. Also, services will frequently overlap in their data requirements, as in Figure 1.1. We use these features to develop a new partitioning strategy that uses just the data required by the services in the current workload.

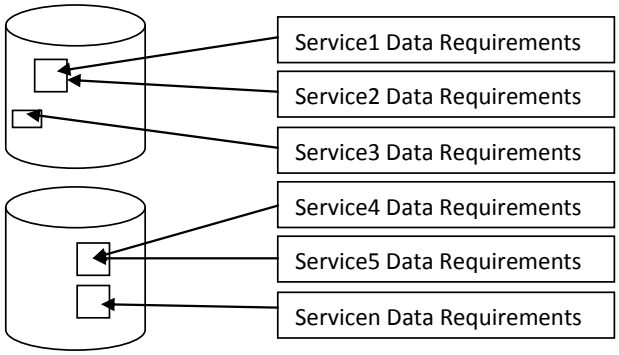

Figure 1.1: Motivating Example for a Partitioning and Allocation System

In recent years, there has been research into partitioning techniques for shorter-running transactions in the Cloud [28] [85]. We seek to partition for conflicting transactions in parallel environments. We do so by introducing an approach that would allow partitioning of small units of data called micropartitions. Micropartitions are allocated in such a way that services and service compositions are executed against a minimum number of database nodes thereby reducing the number of distributed calls executed during a transaction.

Data partitioning has traditionally been used as a strategy for mitigating data access costs. There have been three reasons for partitioning. First, distributed processing environments use partitioning to place data near the processes that need them, thereby reducing transmission costs. Second, partitioning has been used to manage big data, where very large databases are divided and spread across systems to exploit

parallel data access. Third, partitions provide concurrent access for users, spreading the cost of user access in environments where there are heavy user loads.

In the first case, of distributed processing, fragments are designed to provide the data needed for the applications at a particular node on the network. Some form of vertical and/or horizontal partitioning is performed to provide local access to the data. Often, the distribution of the applications is fixed, and the data requirements are analyzed statically. The database partitions are tightly coupled to the applications and must be redesigned for changes to the workload.

In the remaining two cases, of big data and heavy user access, some form of horizontal partitioning is normally used. These partitioning techniques are only effective in reducing costs for a restricted class of applications. In the realm of big data, horizontal partitioning works well for data analysis, tasks where many partitions are accessed simultaneously to calculate statistics. In the area of heavy user access simple requests for data delivery work well against horizontally-partitioned data.

No effective partitioning strategy has been developed, as of yet, for service-oriented environments. Services are reusable, composable, and dynamic. Reusability means that a service can be easily used by different clients, in different contexts (i.e., in conjunction with different services and by accessing data from different underlying database systems). Composability means that services can be combined in different ways in order to meet different business requirements. Dynamism means that services can be discovered in an automated way and can dynamically relocate, assemble, interact with other services. In our work we address partitioning that is responsive to web service compositions.

It is well known that data partitioning doesn't scale well for web services that are transactional in nature. In our work, we develop a partitioning strategy that addresses operation conflicts between web services, which is the fundamental reason why transactional databases do not scale well.

Phase One concerns itself with the partitioning problem, where we seek to partition data in systems that must scale in a cost-effective way. Reducing cost involves minimizing data storage, minimizing the cost of distributed access, and minimizing replication. The first phase of this dissertation contributes to solving the availability and performance problems of using relational databases with services by developing a new partitioning technique.

Our motivation for doing this work comes from the nature of cloud computing, where resources need to scale as needed. Our inspiration came from just-in-time manufacturing, where manufacturers seek to create a highly-responsive production environment by providing only the inventory needed when it is needed [23]. Our contribution is to present a partitioning method that provides minimal data to services executing in a parallel database environment and that permits a web service composition to get all data from a single database node.

Our research is concerned with improving database query response time for system workloads. The overhead costs for workload query processing, in distributed environments, can be very high. The background section describes, in detail, the costs involved in processing queries in a distributed database environment.

For Phase Two, we claim that we can achieve data scheduling for a workload that provides improved performance over traditional scheduling methods. We improve efficiency by creating a schedule that reduces the number of distributed calls due to commits and correctness verification and that selectively supports parallel processing. The second phase of our work shows the scheduling procedure for a multiprocessor, single DBMS node. The scheduling procedure allocates web services from the complex web services, preserving correctness while maximizing parallel execution of the services. We describe an algorithm that creates such a schedule.

Phase Two describes an effective data scheduling method for a mixed workload comprised of complex web services. We use the concept of data similarity developed

in Phase One to aid in the development of the scheduling method.

1.2 Proposed Work

This dissertation is concerned with correct partitioning and scheduling for web service execution in a parallel database environment. The core idea is to design a system in the spirit of just-in-time inventory control where only the data required for current processes are available; it is a fragmentation technique based upon the concept of minimal data sets. We are also interested in developing a scheduling method that that provides efficient concurrency control for web services executing in a parallel environment.

A good example of possible usage of such a system is an enterprise that wants to make all its services, from each department, available for company-wide use. Employees could access and creatively combine available services into compositions that could work against different databases. The database partitioning can be adjusted to reflect an evolving workload. The web service compositions can also be transactional in nature and can be efficiently executed in a parallel processing environment.

We develop a partitioning method based upon clustering of services that use similar data. The data requirements are then defined for clusters of services and partitions are realized from these definitions. We call such a defnition an abstract query.

We develop a graph-based, heuristic scheduling technique for scheduling web service compositions that preserves correctness of data for multi-processor database system. Processes comprised of web services can bring challenges with respect to data consistency. Implementing commit protocols can be costly. Our scheduling technique relies upon use of delays in service execution in order to preserve consistency and avoid use of commit protocols in a parallel execution environment. Our approach to achieving this goal is a two-phased one to provide the efficient data partitioning and scheduling.

Phase I Overview

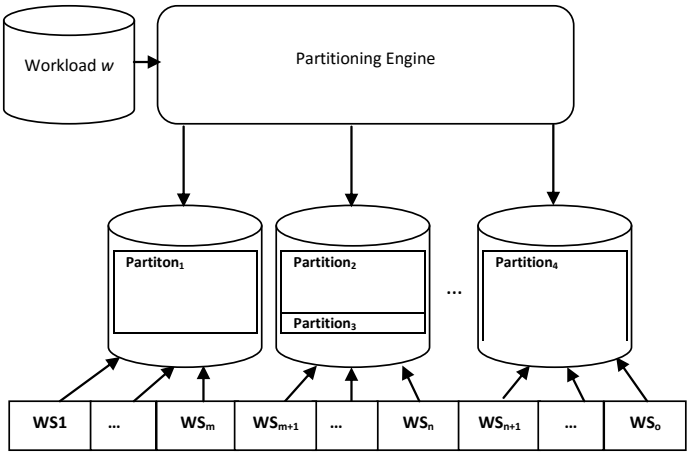

Figure 1.2: Framework for Phase I - Partitioning and Allocation System

Phase I answers the following research questions:

1. How can we efficiently allocate the data to database nodes to satisfy a set of web services having a single query, in a workload?
2. How can we identify and organize atomic services that have similar data requirements?
3. Can we demonstrate that this micropartitioning method is more efficient than the standard range partitioning in use today?

Phase I provides methods to identify data needs of atomic services and then groups services with similar data needs. Abstract queries are created from these groupings. The abstract queries form the basis for micropartitions which are then allocated to nodes. An atomic service is a service that has only one query associated with it.

The Clustering Manager in Figure 1.2 is responsible for grouping similar queries. Given a service inventory residing in an enterprise system, one can extract queries being used by the current workload and group the queries together based on data similarity. We create a hierarchy of these clusters. This hierarchy is updated only when services are added to or deleted from the workload. For each cluster in the hierarchy, we derive an abstract query that represents the data requirements for that group of queries. Each cluster is tagged with the derived abstract query. In the dissertation, we prove the minimality of the data set created from a given abstract query.

The Partition Engine in Figure 1.2 is responsible for selecting a set of clusters from which to create micropartitions. For the given workload, candidate clusters are selected from the cluster tree. The abstract query information stored with the clusters is then used to create the micropartitions that will be stored on the nodes.

In Phase I we assume that all services and nodes are in a central location, as in a cloud. In Phase I we also deal only with atomic services, those services that contain only one query.

Phase II Overview

Phase II answers the following research questions:

1. How can we enhance the partitioning framework to handle a workload comprised of web services with multiple database operations?
2. How can we develop a scheduling method, for a parallel processing environment, that creates cost effective execution sequence of web services with multiple database operations?

In Phase II we develop a scheduling strategy by introducing heuristics that improve performance of web service compositions involving multiple queries. We will call

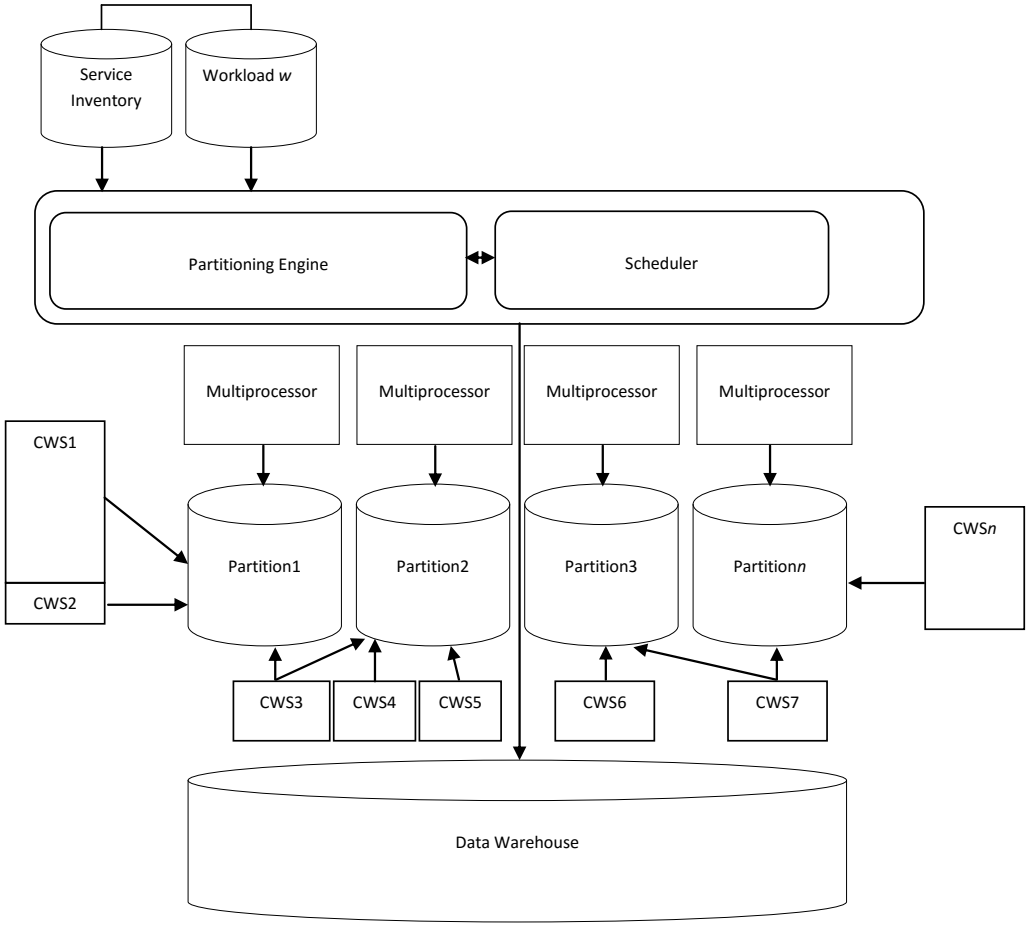

Figure 1.3: Framework for Phase II - Partitioning and Allocation System

these types of services, complex web services, as opposed to atomic web services. The scheduler exploits multiprocessor databases in an efficient way providing concurrency support by ordering complex web services in such a way that delays are minimized due to conflicting operations in complex web services.

1.3 Benefits of Method

Our partitioning technique reduces the cost of executing transactions by localizing data and minimizing data partitions. The partitioning techniques could be extended

to recreate the database partitions automatically so that the system works for dynamically changing workloads. The micropartitioning strategy would be of benefit in distributed environments where fragments and their corresponding services could automatically be organized into sets for dispersal.

The scheduler presents a new way of handling concurrency control in a multi-core environment. It has become apparent that concurrency control methods developed for relational database systems do not scale well with in-memory multi-core database systems. The scheduling approach is an attempt to avoid the bottlenecks created by other concurrency control techniques that are commonly implemented.

1.4 Dissertation Outline

The subsequent dissertation is outlined as follows: Chapter 2, the background section, discusses related work in data partitioning and scheduling, web services, clustering and allocation algorithms. Chapter 3 presents Phase I which is clustering and allocation of atomic queries, presents an implementation and discusses implementation results. Chapter 4 presents Phase II which defines complex web services, quantifies delays imposed upon complex web services when ordered in a schedule, and uses the delay information to efficiently order complex web services for execution in a parallel database environment. Chapter 5 presents our conclusions.

Chapter 2

Related Work

Service-oriented architecture and distributed storage have contributed to the advancement of cloud computing. Because of high and variable usage requirements and high data storage requirements, traditional methods for managing software and data are insufficient for the cloud. Much research has been devoted to enabling software and data usage in such an environment. In this section, we present an overview of just-in-time manufacturing techniques and of service-oriented architecture. We present developments in distributed computing, database partitioning, and concurrency control. We present an overview of algorithmic techniques used in the research.

2.1 Just-In-Time Processing

Massive data storage requirements and high levels of user database processing require new management techniques. How we manage this new environment has been the subject of research in recent years. The history of manufacturing could provide valuable lessons in process improvement for the database community.

Lean manufacturing is a refined manufacturing approach that seeks to create more responsive systems. A lean system quickly responds to customer demand but does not hold on to excess inventory in order to do so. A lean system keeps a minimal amount of inventory on hand and depends upon good supplier communications and transportation to ensure timely production. A lean system is also designed to quickly retool to produce different products from a line of products.

The just-in-time, JIT, philosophy of production emphasizes minimizing the amount of on-hand resources used in enterprise production activities. JIT, or lean manufacturing, focuses on paring inventory to only the minimal stock needed for current operations [23]. Instead of being overstocked in order to manage any potential issues, JIT shops focus on improving the connection between the supplier and the factory and between the factory and the customer. By reducing inventory, an operation can save money. Also, by reducing inventory, inefficiencies in the system are exposed and corrected.

Just-in-time includes a set of principles and practices to reduce cost through removal of waste and through the subsequent simplification of all manufacturing and support processes. In reality, reducing inventory is usually performed by a combination of inventory-by-forecasting and just-in-time inventory analysis. In this work, we address the development of a data partitioning technique that combines just-in-time data provisioning with predictive processes to support just-in-time processing. Just-in-time manufacturing identifies muda, or seven forms of waste [67].

1. Inventory - all components not being processed
2. Overproduction - production that is ahead of demand
3. Transportation - moving products that are not actually required to perform current processing
4. Motion - people or equipment moving more than necessary to perform the processing
5. Waiting - waiting for the next production step
6. Overprocessing - any non-value added activity or component that the customer would be unwilling to pay for
7. Defects - the effort involved in inspecting and fixing defects

The primary form of waste is excess inventory. This comes from storing raw materials not immediately needed and producing product not currently requested by the consumer. Manufacturers in lean environments seek to minimize the amount of materials and products that are stored on site and, instead, cultivate good communication and transportation lines between themselves and suppliers. Suppliers may, in turn, push back their inventory requirements to suppliers further up the chain.

In this dissertation, we apply lean principles to the storage and access of data. First, with respect to data storage, we apply the principle of inventory reduction by analyzing the minimal amount of data required for transactions in a workload to execute successfully and provide just that minimal amount of data in front-end databases.

Second, transaction costs increase because of the high overhead associated with maintaining correctness of data in environments of increasing parallelism. In fully distributed parallel execution environments, the costs of concurrency control are high because of the requirement of maintaining global concurrency control. In multiprocessor, shared memory parallel executing environments, current concurrency control mechanisms are costly because of the bottlenecks the existing protocols create.

Therefore, in a no-share environment with many DBMS nodes, there is the intuition that concurrency control mechanisms lead to overprocessing distributed transactions, contributing to slower response times. In such environments there are high costs associated with maintaining concurrency control .

2.2 Service Oriented Architecture

Service-oriented Architecture, SOA, permits software initially designed to be used within a limited scope of operation to interface with units of processing written in different languages residing on different platforms. The basic functional gain of any

such architecture is to allow software units to interoperate that were not initially capable of doing so. Such an architectural approach promotes reuse and encourages combining software to solve new and varied computing problems [44].

Any service-oriented architecture implementation must provide a means of making disparate software interoperable. The architecture would provide application programming interfaces for use by the developer. The software itself remains a black box to the client, who knows only the functionality from the interface and no detail. The developer uses standardized design principles to integrate the needed software. Any service-oriented architecture also creates a federated store of software for assembly and reuse.

To date, composition of services has been most heavily used in grid environments. Managing data for workflows in grid environments has had research focus. Researchers analyze the data management requirements of three service-based grid systems [86]. A widely-used data grid management system, DGMS, is Storage Resource Broker [91]. However, current workflows are generally statically defined and data is allocated accordingly.

There is active research on service composition and orchestration. There is research in developing systems that dynamically bind services to a composition at runtime and also heal any runtime violations [116]. In other work, the compositions are automatically or semi-automatically assembled from sets of candidate services to fulfill the users requirements. Most of these methods are inspired by cross-enterprise workflow research and AI planning research [90].

Dynamic workflow methods have inspired automated web service composition. AI planning and deductive theorem proving techniques have been used to automate compositions. AI systems evaluate preconditions and effects that have been placed in service documentation files to automatically determine which sets of services will accomplish a task. Systems have been developed to automatically select and or-

chestrate services [87]. Until recently, scheduling of web service compositions has been a small component in the research papers on automating web service compositions [110] [115] [36]. There has been recent work in the efficient scheduling of web service compositions in order to take advantage of parallel execution. Everest is a multi-tenant scheduling service with future work addressing dynamic scheduling of compositions [100].

Service compositions are also executed in cloud environments or can be designed to execute across several clouds and enterprise systems [71] [66]. To our knowledge, data partitioning for service compositions has not been addressed.

Reusability means that a service can be easily used by different clients in different contexts, where the service accesses data from different underlying database systems and the service is used in conjunction with different services. In order to support reuse any SOA implementation ensures that the unit of logic that is the service is designed to be easily reusable. The logic therein is carefully designed with a thorough understanding of the business context in which it resides and potential areas for reuse. Since multiple users would be accessing the services, reliability becomes a key issue in service design as well.

In order to enhance the reusability feature, a service should also be able to access heterogeneous data sources and not just be linked to one database or even one type of database. Researchers in the field of scientific computing have developed approaches to interface their services with different underlying databases within the context of the web services implementation. The specifications for this approach are called Service Data Objects or SDO [93]. OASIS has also developed the SDO which is a standard to manage access to heterogeneous data sources [14]. Apache Tuscany Project, which is a service-oriented infrastructure, uses this approach for data access [40].

The WS-DAI, Web Services Data Access and Integration Specification, provides a

set of base specifications for creating service-based interfaces to data resources, that other services or applications could, in turn, call [10]. WS-DAIR is an additional set of specifications for defining service-based access to relational databases [105]. Additional standards for data access have been developed for non-relational data stores as well [9] [11]. Grid systems have adopted the above standards in order to access and integrate structured data [108].

The point of creating a service paradigm is to develop systems where services can be quickly found, used, and reused in new and varied situations. The service-orientation seeks to maximally exploit existing applications by making them available to a large audience, by making them easily operable in varied environments, and by making them easily interoperable with other services and with different data sources. As a paradigm it has been available for more than a decade, implemented using XML, and is supported by many standards set by industry leaders. Automation of service discovery and service composition facilitate service usage. Easy coupling of services to varied data sources also increases a service's utility.

The focus of research in the area of discovery is to provide adequate and thorough semantic annotation, for both functional and non-functional aspects of a service. Therefore, researchers seek to create documents with more complete, semantic annotations so that systems can easily and precisely discover the best candidates from a service library [81]. Another research concern is the full automation of service discovery [38].

Service-oriented architecture is one of the foundational technologies of clouds. Cloud infrastructure is virtual and dynamic, with resources appearing and disappearing as needed. Because of this environment no state should be held within the resource. State is held within the representation of the resource.

Service-oriented architectures are an intermediate step between the older generation of distributed computing technologies and current cloud computing initiatives.

The ideal of SOA is to have a clean partitioning of functionality and a constant representation of those services, which is currently the RESTful service implementation. RESTful services is an architectural style to build lighweight, maintainable, and scalable web services. Cloud computing has accepted this service paradigm for effective deployment of services for clients.

Although web services is a mature technology there is considerable effort to continue to advance the technology by making the APIs open. This would achieve the end of making services more available and would allow a client to use services from different cloud vendors and to more easily port services from one cloud to another [111].

In Figure 2.1 we have a layered representation of the administrative costs involved for processing the service and database requests of a workload.

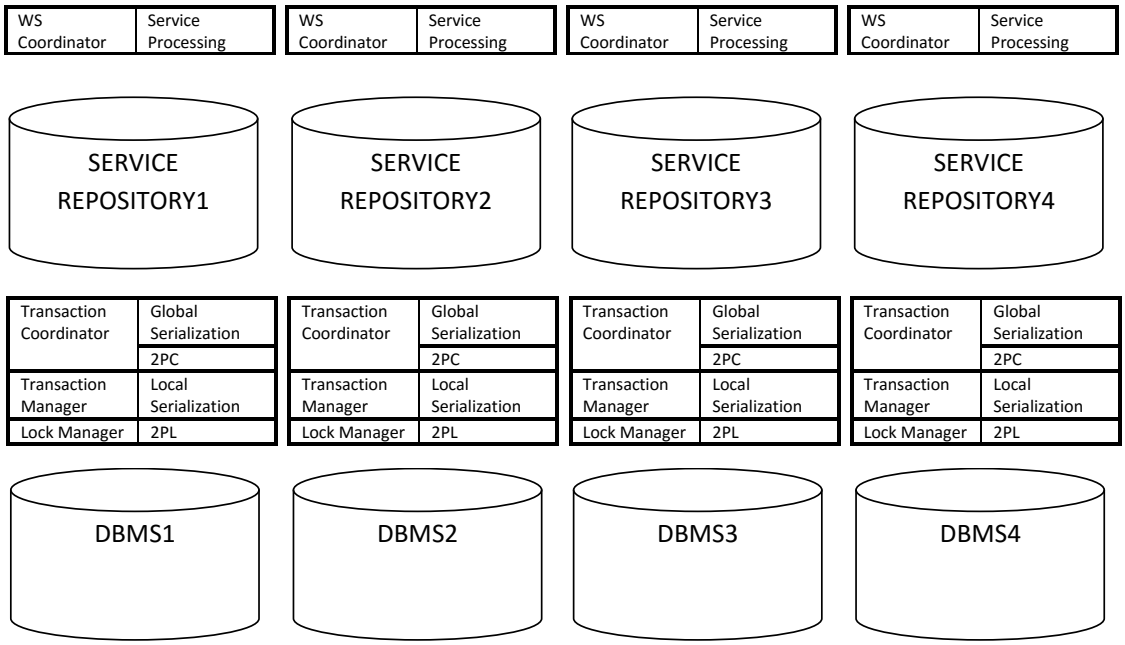

Figure 2.1: *CWS* Transaction Processing Costs

The WS Coordinator manages the execution sequence of web service within a web service composition. This is typically done by executing a flow sequence described in an orchestration document. Such documents are developed using an orchestration language such as BPEL. Messaging is the method by which data is transmitted from one web service to another web service; such messaging of data between web services can be costly when done between application nodes. Our partitioning method localizes the set of services in composition, so that web services that pass data are found on the same application node, thereby reducing transmission cost.

Each DBMS node has a transaction coordinator. Each transaction coordinator has a direct interface between itself and the other transaction coordinators in a distributed database system. If a transaction involves several databases, the participating transaction coordinators must implement two-phase commit protocol across database nodes to ensure correctness of transaction results. Our partitioning method selectively places data for web service compositions onto a minimum number of database nodes, thereby reducing the two-phase commit costs.

The transaction coordinator also handles distributed serialization of operations. This can be very costly. When the data is localized for web services executing within a web service composition, the serialization process is also localized. The transaction manager then handles most serialization of operations locally. Our partitioning method shifts the serialization process from the transaction coordinator to the transaction manager.

Distributed lock managers are employed in transaction execution in order to ensure consistency of data when replicated data must be updated. Distributed locking costs are reduced when data replication is tightly controlled.

A significant, outstanding research issue in cloud environments is the cost of query execution against the underlying database structure. Initially, when optimizing the database for a particular workload, performance will be excellent. But, due to the

dynamic nature of such an environment, with a changing user base and changing workloads, the problems of less than optimal performance, due to mounting data shipping costs and data and access skew increase. Researchers have explored different ways of easing the mounting performance problems. First, new partitioning strategies have been developed to reduce the frequency of distributed transactions, reducing data shipping costs. Second, alternate concurrency control mechanisms have been explored, namely, speculative locking, multi-versioning, and optimistic concurrency control [92] [17] [45] [62] [65].

2.3 Distributed Computing and Storage

Three technologies have contributed to the advancement of cloud computing, virtualization, service-oriented architecture, and distributed computing and storage. For our research, we do not address virtualization technologies. We have addressed SOA and in this section we review distributed computing and storage technologies.

In the future, there will be easier availability of many more services to use and to orchestrate with other services in the cloud. Open APIs for services are available on different vendor clouds. Grid computing is similar to cloud computing, but there major challenges before grid applications can be ported over to clouds. Grid computing is typically linked to science where scientific applications require highly-distributed computation. Grid computing generally uses a highly heterogeneous system of computers and databases to perform highly-distributed analytics which are then consolidated. A key property of grid computing which cannot be satisfied by cloud computing is federation. The basis of grid computing is virtual organization, which manages resource sharing, user authentication and access control. Grid computing also requires high performance and high-scalability. Although cloud environments provide high-scalability they do not provide high-performance environments at the level needed by

grid computing.

Databases have been dominated by the relational model for the past several decades. Opinion is that the rigid structure of relational databases, although handling complex and large amounts of data well, is not suitable for a cloud environment. Rigid can be interpreted as difficulty altering schemas and difficulty scaling out.

We review the processes and costs involved in processing a transaction against a database consisting of loosely-coupled sites where each DBMS system is independent of the others and the transactions may access one or more DBMS nodes [83] [96]. The DBMS systems involved may be homogeneous. DBMS node homogeneity means that all nodes have identical software. All nodes are aware of one another and cooperate to execute user requests. Nodes give up some of their autonomy in that they cannot independently change software or schemas.

The DBMS systems involved may be heterogeneous in nature, in that, different sites use different software and schemas. Such differences in software and schemas pose major problems for distributed transaction processing [54]. Nodes may not be aware of one another or may provide limited capability in cooperatively processing transactions.

Distributed data storage, in the relational model involves replication. Systems maintain multiple copies of data at different locations for the purposes of fault tolerance and faster retrieval. Distributed data storage also involves fragmentation, where relations are partitioned into several fragments stored at different nodes. A distributed data storage system combines fragmentation and replication in order to improve efficiency. Generally, relations are fragmented and several copies of the fragments are maintained at different nodes. Full replication means that a relation is copied, in its entirety, to another node. Fully redundant databases means that the entire DBMS is replicated to another node.

Replication is advantageous because it provides availability, parallelism, and re-

duced data transfer. If a node containing a relation should fail, other nodes would have a copy of the relation making it available for processing. Queries may processed against several replicas of data simultaneously. Local copies, made available for use, reduce data transfer costs.The disadvantages of having replicas is that updating copies can be costly. Also, the mechanisms for keeping all concurrent data consistent can be quite complex. One mechanism for concurrency control is to make one replica the primary copy and apply all update operations to it. This is the primary copy method protocol.

When queries are processed against distributed data there must be transparency for the user. The user is unaware as to whether they are accessing a fragment, a replica, or a particular location. In order to implement such transparency every data item in the system should have a unique name. There should be an efficient protocol for finding a data item. It should be possible to switch locations when accessing a data item without disruption of processing. Each node should be able to create new data items independently.

For this naming procedure and location procedure there are two common methods of implementation. The first method of implementation is to have a centralized server called a name server which, names all data items. Nodes maintain names for all local data items. Name servers maintain names for all non-local data items. This method allows for efficient and transparent access to data. It does not, however, allow local nodes to name their own data items. The name server can also be a bottleneck to processing.

The second naming method is the assignment of aliases. Each node prefixes the data item name with a unique node identifier. Each data item, at each node, has the set of all aliases stored with it so that the data item can be transparently retrieved from any node. A distributed transaction may access data at several sites. Each site will have both a transaction manager and a transaction coordinator.

As Figure 2.1 shows, the transaction manager performs local transaction administration. It performs transaction logging for recovery purposes and concurrency control. The transaction coordinator is responsible for processing a query in a distributed way, if all data is not available at the local node. The transaction coordinator starts the transaction. The transaction coordinator creates subtransactions that are then sent to the appropriate sites for execution. The transaction coordinator then supervises the termination of the distributed transaction. The transaction coordinator will either successfully terminate the distributed transaction or will supervise the rollback of all data to a previously consistent state should an error occur during the transaction process.

Each node has both transaction coordinator and a transaction manager. The coordinator manages transactions in conjunction with the relevant transaction managers.

A distributed transaction can fail if there is a DBMS node failure, if a communications link fails, or if a message is dropped. Distributed transaction processing is more fragile than localized transaction processing. Commit protocols are introduced into the transaction coordinator software to ensure that all DBMS remain in a consistent state despite any system failures that may occur. Either a transaction succeeds completely or, because of some failure in the process, the transaction is rolled back completely. A commit protocol ensures such atomicity in a distributed environment.

The two-phase commit protocol, 2PC, is commonly used. The three-phase commit, 3PC, protocol is more complicated and expensive but provides additional protections over and above the 2PC. 2PC assumes a fail-stop model. The failed nodes simply stop working and do no more harm such as sending additional messaging after the failure. The 2PC protocol is initiated by the transaction coordinator after the last step of the transaction has been completed.

In the first phase of the commit protocol, the transaction coordinator asks all par-

ticipating nodes if they are prepared to commit the transaction. This is the prepare phase. Upon receiving the message, the transaction manager at each node determines if it can commit the transaction and replies to the transaction coordinator with a commit or an abort response.

In the second phase of the commit protocol a record is made of the transaction managers' decisions. If any transaction manager aborts then the transaction coordinator broadcasts an abort message to all participants. If all transaction managers are ready to commit, the transaction coordinator broadcasts a commit message to all participants. The participants then take appropriate action locally.

If the transaction coordinator node fails then the transaction is blocked until the node is up and running again. Once the coordinator node is up and running again, one of several courses of recovery may be initiated. If the coordinator node's log indicates that the distributed transaction is ready to commit, the node redoes the execution of the transaction. If log contains an abort record the coordinator site aborts the transaction. If log contains a ready message, the node must consult the transaction coordinator as to the fate of the transaction.

Conflicting transactions are those where there is a conflict between operations from each transaction. The standard read-write conflict between two queries states that one query reads a data item and the other query writes the data item. For example, there is a potential conflict where query one reads a data item, query two writes the same data item and commits, and query one reads the data item again. Query one has read the data item twice, but the data item is not in a consistent state. This is otherwise known as an unrepeatable read.

The standard write-write conflict between two queries states that both queries write to the same data item. If either query rolls back its write there is a potential that the committed transaction will have stored incorrect data. This is known as overwriting uncommitted data. The standard write-read conflict between two queries

states that one query writes to a data item and the other reads the data item. There is a potential conflict in that the read query is reading uncommitted data. This is otherwise known as a dirty read and if the write query rolls back its write for any reason, the read query will have computed on incorrect data.

When there are such conflicts between queries, serializability enforcement techniques must be employed to ensure that a schedule of interleaved read and write operations from different queries results in the same outcome as a serial schedule of those queries. Such enforcement techniques can be precedence graph cycle elimination, SS2PL, Strict Strong Two-Phase Locking, or 2PL, Two-Phase Locking, timestamp ordering, or snapshot isolation. When such transactional integrity enforcement techniques must be realized in a distributed environment, the additional overhead of passing lock information, for example, from one database node to another, can become very high. Thus, the cost of ensuring serializability at the global level, as in the cloud or other distributed environment, can be very high.

Concurrent transactions are executed in a preemptive, time-shared method. Context-switching creates an interleaved schedule and concurrency control methods ensure that the interleaving is proper in that it results in correct and consistent database updates. Conflict serializability scheduling detects and specifies the correct order for two conflicting operations. A view serializable schedule allows all conflict serializable schedules plus blind writes. Conflict serializability and view serializability are techniques that were developed to address the development of schedules for a single processor where multiple transactions could be interleaved, resulting in a balancing of the execution times for the set of transactions.

One common architecture for a multi-processor environment is a parallel database system. A parallel database can be a single database machine with multiple processors. It can also be a number of database nodes that are physically close to one another, connected by high-speed communications lines thus assuming that the com-

munications cost is small. It can be a shared memory system, a shared disk system, or a no-share system.

The common architectures for a parallel database system are shared-memory and no-share. In shared-memory, all processors access the same memory and disks. This is standard and accepted architecture today. Shared memory architecture has processors accessing the same global address space. For a no-share environment, every processor has its own machine and environment. In this distributed-memory architecture processors only have access to their respective local address spaces.

There is flexibility inherent in a shared-memory architecture in that it can simulate a distributed memory architecture by dividing its global memory into disjoint parts and assigning these disjoint parts to the different processors. Shared memory is good from the vantage point that communication costs do not need to be taken into account in parallel execution of tasks and load balancing is more-easily managed in this type of environment. However, there are scalability issues associated with a shared-memory architecture due to increasing points of contention in memory and cache coherence as processors are added. Inter-transaction parallelism is easy on a shared-memory system, which guarantees serial execution of a single transaction. The enforcement of serializability is expensive, however, with long duration waits and a high number of aborts. However, concurrency control mechanisms, such as locking, are limiting factors on scalability for multi-core database systems. Scalability on multicore systems can be hampered by even a single point of contention [46].

Current methods for implementing concurrency control, regardless the mechanism, whether it is two-phase locking, basic timestamp ordering, multiversion timestamp ordering, optimistic concurrency control, or timestamp ordering with partition level locking, do not scale well on multi-core systems [97].

Distributed database, on the other hand, are characterized by high communication costs for communicating between nodes and is usually a shared-nothing architecture.

Maintaining concurrency control for a parallel database system and for distributed database systems require additional overhead and the costs for a distributed database system are even more than for a parallel database system, given the communications overhead.

2.4 Partitioning for Distributed Database Systems

Database fragmentation is a method of dividing larger database systems into smaller sections in order to improve performance, availability, and security. The foundations of database partitioning, or fragmentation, adhere to the basic definition of set partitioning. Viewing a relation in an RDBMS, one can partition the relation vertically, horizontally, or both. The correctness rules for database fragmentation are completeness, reconstruction and disjointedness [83]. Each rule corresponds to the conditions defining mathematical partitioning of sets.

1. Decomposition of relation R into fragments $R_1, R_2, ..., R_n$ is complete if and only if each data item in R can also be found in some R_i.
2. If relation R is decomposed into fragments $R_1, R_2, ..., R_n$, then there exists some relational operator $\bigtriangledown$ that reconstructs R from its fragments (i.e., $R = R_1 \bigtriangledown ... \bigtriangledown R_n$. The relational operator $\bigtriangledown$ is union for horizontal fragments and join for vertical fragments.
3. If relation R is decomposed into fragments $R_1, R_2, ..., R_n$ and data item d_i appears in fragment R_j, then d_i should not appear in any other fragment R_k, $j \neq k$. For horizontal fragmentation the data item is a tuple. For vertical fragmentation the data item is an attribute.

In standard distributed database systems, the databases are fragmented based on

queries in applications found at specified locations. The fragments are then placed and replicated at the applications' location. The frequency and nature of user accesses also influence the replication and allocation of the fragments. Many distributed database systems use fragmentation, replication, and allocation methods to achieve high performance or data access throughput. Replication of the fragments can be performed to increase availability and can be modeled as partial or full replication. The allocation/replication process can be modeled as a cost function under a set of constraints.

Many researchers have studied fragmentation and allocation for DDBMSs. Most methods provide heuristics for fragmentation because optimal solutions are too costly [83]. A grouping technique for fragmentation in distributed database systems was developed and a splitting technique [95] [79]. Heuristic solutions for fragmentation and allocation of data have also been developed [83]. A number of approaches have been developed for analyzing the cost of data placement and for finding methods to minimize that cost [74] [12] [68]. Other research in wide-area networks examined data allocation in a larger network setting [55].

Database systems, other than the traditional RDBMS, have been developed over the past decade, as a response to new use cases that RDBMS, as currently approached, cannot easily satisfy. We have seen the development of data warehouses and noSQL databases.

Data warehouses store large amounts of data for information evaluation and analytics. Data warehouses are much larger than traditional transactional databases and support analysis rather than transactions [22]. Standard partitioning techniques have been used to manage the volume of data in warehouses [107][53]. Partitioning methods for the specialized star schemas of data warehouses have also been developed [15] [33].

Web applications interact with very large NoSQL data stores. Each commercial

NoSQL store addresses a different set of use cases. The partitioning strategy for such systems involves scaling out data onto many commodity machines using a horizontal partitioning strategy such as hash, list, and/or range indexing. Partitioning is an integral component of such massively parallel systems [20]. The partitions are called shards and each shard is under the control of a separate data management system.

The web partitioning strategy works well for a limited class of applications that require information retrieval. There are several commercial products out on the market today that use some form of sharding to support data access. Requests for data content or statistics can be farmed out to all machines simultaneously. The MapReduce paradigm has been successfully exploited to perform simultaneous requests against multiple nodes housing data shards [34]. There is also a high-level of replication of each shard to ensure availability and fault-tolerance.

Data warehousing systems also work well with data sharding. Data in dimension tables is horizontally partitioned and disseminated to different nodes in a no-share system. Queries against data warehouses are often ad hoc, complex, read-only queries that produce summary statistics. These type of queries can be decomposed and sent to separate partitions, and intra-query parallelism can be exploited to get intermediate results which can then be combined. These operations can all be performed without internode communication.

C-Store is a data warehousing system that horizontally shards data across independent DBMSs [99]. The authors explicitly acknowledge that any new DBMS architecture, of which many have appeared in the last few years, should assume a massively parallel environment. This means that new DBMSs will be systems with many independently operating nodes, such as clouds, with homogeneous commodity machines, or grids, with heterogeneous commodity nodes. The nodes may be physically co-located or not. The authors state that there will be potentially many nodes in such systems so allocation of data to the nodes must be done in an automated

fashion.

C-Store is a column-store database, like many of its data-warehousing predecessors. It uses techniques such as overlapping materialized views to serve data. These overlapping materialized views, as a whole, cover the entire database. The views are connected by join indexes whereby the entire database can be recovered. The overlapping materialized views can be sharded across nodes for faster response time. Vertica is the commercial offshoot of CStore [4]. The C-Store architecture is optimized for an analytic workload. The researchers also tried to accommodate transactional workloads by separating write-operands into a write-optimized store, called WS, and read-only data into a read-optimized store called RS. Data in WS is moved to the RS in a controlled fashion by a tuple-mover. However, the researchers acknowledge that this system can only give reasonable response time for transactional workloads. C-Store, then, achieves very high performance on analytical queries and reasonable performance on transactions executed in an on-line environment, or OLTP (online transaction processing) environments.

Because certain web applications and data warehousing systems work well in massively parallel systems, these applications have been the first to migrate to enterprise and internet cloud systems. There has been more difficulty in migrating transactional systems to clouds. The primary stumbling block has been the provision of good scalability while preserving the traditional ACID properties of standard transactional systems. Generally, one of the ACID properties, atomicity, consistency, isolation, or durability, has had to be sacrificed, in order to provide scalability and, by extension, adequate reliability and response-time that users require. Developing good partitioning and allocation strategies for transactional databases when dealing with large data and large user volumes has been a major issue [48].

NewSQL databases are relational databases that would provide high scalability and would preserve ACID properties for transactional workloads [52]. NewSQL

databases are databases that adhere to the relational model but that have architectures that make them perform better in systems that must scale. NewSQL databases also seek to retain SQL declarative language in order to perform queries. The target workload for NewSQL databases is comprised of many transactions that are short-lived, that touch smalls subsets of data using index lookups, and that are repetitive, that use the same queries with different inputs.

Sharding can put an internet infrastructure into deep freeze [25]. The problems with sharding are that the shards are carefully designed for a set of mutually compatible transactions, transactions that perform tuple selection on conditions using the same attributes. The sharding is engineered precisely along the attributes that participate in the projection statement. Once new queries, in transactions, are introduced that may not access tuples along the same attributes, overhead increases exponentially because tables that need to be joined are on different shards, or tuples that need to be accessed are in different shards. The task of accommodating new queries must be coded into the associated application logic which becomes difficult for the application developer. Therefore, shards cannot accommodate heterogeneous workloads, that is, workloads that incorporate queries using conditional statements with different attribute values. Shards can, therefore, cannot accommodate changing workloads without encountering development difficulties.

HStore uses a partitioning strategy that exploits elements of microsharding and graph-based partitioning [85] [103]. Horizontica was the first commercial offshoot of HStore [2]. VoltDB is the latest commercial offshoot of HStore [5]. EStore furthers the research efforts of HStore, allocating data for certain queries in the workload on a tuple-by-tuple basis. Microsharding is another strategy meant to partition databases according to analyses performed on OLTP workloads . Elements of microsharding are already implemented in Google App Engine, GAE, and Google's Megastore in that small partitions, called entities are created that represent a logical

unit in the data store. These entities are then collected into entity groups. The idea has been extended to relational databases and a small partition is created, called a microshard, that would provide data for an entire transaction; the microshard is created from a definition called a transaction class [104]. Transaction classes contain information about the dataset required. However, these transaction classes are created manually. Again, microsharding addresses the transactional nature of the activities but does not address reusability and dynamism.

Relational Cloud is a system developed to provide Database-As-A-Service in the cloud [3]. As such, it was a research effort determined to provide transactional processing in a cloud environment with ACID guarantees. Relational Cloud uses the Schism partitioning system and uses workload information to guide the partitioning and replication strategy [28]. The focus is to handle transactions within as few partitions as possible to reduce the cost of two-phase commits. The first part of the Schism partitioning process involves building a graph from SQL trace workload information. A graph is built where the nodes are tuples accessed by the workload. An edge is introduced where two tuples are involved in the same transaction. The graph is then partitioned between the nodes with the fewest edges in order to isolate tuples that belong together for transactions. The actual database partitions are derived from this structure. The graph-based partitioning strategy is useful for creating accurate partitioning so that transactions are performed in only one partition and are therefore cost-effective. However, this strategy is NP-complete. The authors introduce heuristics to speed partition creation but at the expense of accuracy. Our strategy creates micropartition definitions based upon the database schema information rather than using a graph-based approach at the tuple level. In our approach, the fragments are allocated to minimize the cost of transactions as well. Schism also only horizontally partitions the data while our process partitions both horizontally and vertically.

HStore uses a skew-aware partitioning system that also seeks to limit transac-

tions to as few nodes as possible while also seeking to balance processing load among machines. HStore also uses a graph-based partitioning scheme but each node is a table and each edge is a join in a query. Partitions are created off the graph in a manner similar to Relational Cloud, but entire tables are migrated to different nodes. Seemingly, HStore partitions can be more quickly redefined, in the reconstruction of the graph, based on changing workloads, as compared with Relational Cloud. However, the partitions are still treated as an entire unit and, therefore, their physical redefinition and relocation would not be responsive.

An open source database called Oinky uses relational microsharding to manage data [63]. Here, the database is divided into very small partitions of logically, related data. The application that needs the data will load only the microshards it needs into memory and will perform entire transactions against the data. When the transaction is complete, the data is then written out to database storage. Thus, the system handles atomicity, consistency, and isolation. It is unclear, however, how the overhead costs of loading into memory and writing out to disk have been taken into account. Again, the division into small partitions is statically-defined. The focus has been to maximize the efficiency of transactions.

To date, database partitioning has been handled in an offline manner, where a database administrator tunes and partitions the database manually to closely match the queries in the workload in order to optimize performance. There have been some attempts to automate the partitioning process, in order to repartition as the workload mix changes or the database change size or user access spikes [59] [6].

The search space of possible partitions is large, making it, as stated before, infeasable to always find the optimal solution. However, good solutions are feasible for automated partitioning, as good solutions have been feasible for automation of query planning. There are many algorithms for developing partitioning strategies offline [78] [95]. They cannot be feasibly used in an online setting.

AUTOSTORE is an attempt to automate vertical and horizontal partitioning by representing partitioning, in general, as a one-dimensional partitioning problem and both vertical and horizontal partitioning as subproblems of 1DPP (One-Dimensional Partitioning Problem). The online database partitioning algorithm O^2P solves 1DPP. O^2P avoids the brute-force method of enumerating all possible splits in the 1DPP [59].

2.5 Concurrency Control for Distributed Database Systems

For the cloud environment, building stateless services required rethinking database consistency. Traditionally databases have been ACID-compliant, focusing on ensuring that a database is consistent and complete. ACID transactions use some form of locking mechanism while updating database information and achieving a consistent state. In an ACID system the database cannot be left in an inconsistent state. This is an issue in high-volume, high-demand systems, such as a cloud environment, where many users and many services are competing for the same resource. The delay incurred by locking a database momentarily, to ensure consistency of all the data, becomes intolerable if there is a high number of users and a high number of services accessing the same database.

BASE-compliant, transactional database systems are the current solution to this problem [88]. BASE is focused on ensuring that resources are always available and that data eventually will become consistent. The database does not lock, ensuring that users and services always have access to the data. However, this approach generates a percentage of transactions that must be reconciled or may fail. This is handled by developers by audits, retries, and reconciliation processes. This is harder to program but does allow for high scalability and good performance.

The problem, however, is that it becomes difficult to port standard RDBMS and their associated applications to the cloud without having to make major alterations

to the systems. To transfer an ACID based application and the database to the cloud without making those major changes, one has to deploy the application to a single partition, losing much of the value of the cloud because the application cannot scale.

A relational database system cannot scale well, as explained by Brewer's CAP Theorem [51]. The CAP Theorem proves by contradiction that there cannot be a guarantee of both availability and atomic consistency in an asynchronous network model implementing a read/write data object. In a partially synchronous environment most of the data can be reliably returned most of the time. It is impossible to provide reliably consistent data in a partitioned network. In other words, the CAP Theorem states that only two out of three requirements can be guaranteed in such a system.

1. Consistency - All nodes see the same data at the same time. Any read operation that starts after a write operation should see the data that was written, which is atomic consistency. Multisite transactions have the all-or-nothing semantics supported by current DBMSs. All replicated data are in consistent states.
2. Availability - Every request receives a response about whether it succeeded or failed. Every request to a non-failing node in the system must result in a response. A DBMS must always be up, switching over to a replica if a failure occurs.
3. Partition tolerance - Despite system failure or message loss, the system continues to operate. When a network is partitioned messages cannot be sent from one node to another in the system. No set of failures less than total network failure results in partition intolerance. Processing continues even if there is a network failure that splits processing nodes.

Because CAP states that only two of three non-functional requirements can be sat-

isfied, this implies that there are only three types of distributed database systems that one can build, consistent-available, consistent-partition tolerant, or available-partition tolerant for distributed environments that must scale.

In designing cloud databases, there has been a requirement to select one of the three previously mentioned options. The selection of availability and partition tolerance has been the focus for the new NoSQL database systems. Typically, NoSQL systems do not allow transactions to cross boundaries. Therefore, consistency is limited to making sure that all replicas have the same data. NoSQL developers have found it acceptable to drop the consistency requirement, replacing consistency with eventual consistency. NoSQL databases satisfy web services which are expected to be highly available. Even the smallest delay in response time can lead to significant losses in a customer base. The goal of most web services is to respond with maximum speed of the network. Partition tolerance, as a type of fault tolerance, is also a requirement for current web services.

Other distributed database systems advocate the selection of availability and consistency claiming that network partitions are rare especially when network connections are replicated [98]. Several new database systems have chosen the availability and consistency requirements over partition-tolerance. There is also a community that states that new database systems should not be based solely on the CAP Theorem claiming that it is insufficient to explain the engineering tradeoffs in a distributed database environment [1].

There is the view that latency is a more important issue than partition tolerance in analysis of distributed systems. Latency is not taken into consideration when performing the CAP analysis where latency can be seen as ever-present, whether there is a network partition or not. Unavailability due to partitioning can be seen as latency beyond a certain point in time. Reduced consistency is attributable to latency, not to partitioning. The issue then becomes how to handle the tradeoff between latency

and consistency. Because latency occurs when there is replication in the system the important issue becomes to manage replication.

There are three types of replication, replication where all nodes are updated at the same time, update is sent to a master node first, or update is sent to an arbitrary node. When all replicants are updated at the same time, update can be done with or without a preprocessing or protocol execution step. Without the additional coordination step, the strict definition of consistency is lost. The coordination step, however, adds latency. In the second case, where a master node receives the update first and coordinates the update to all other replicas, the update process for the slaves can be done in a synchronous or asynchronous manner or a mix of the two. Synchronous update would ensure consistency but introduce latency. Depending on how asynchronous communication is handled, one could have inconsistency or latency or both. The third type of replication update has an arbitrary node receiving the update, where extra latency may occur from resolving the location of the update. In any case, with replication there is always a tradeoff between consistency and latency.

CAP actually states that when there is no partition one can achieve both consistency and availability. It is only in the presence of a partition that one must make the choice. CAP, therefore, is only an explanation for the failure scenario. Therefore, the CAP Theorem does not really justify the choice to automatically reduce consistency guarantees in the NoSQL database systems. These systems also may reduce the other ACID guarantees as well [1]. CAP, therefore, is only one of two major reasons for reduced consistency in distributed DBMS environments. It is important to attend to the consistency and latency tradeoff because that one is constantly present, whereas the consistency and availability tradeoff only occurs in the relatively rare scenario of a network partition. The new formulation PACELC handles the tradeoff of consistency with latency making the CAP Theorem more limited than originally perceived [19].

There are fully ACID compliant DDBMS systems, such as VoltDB, HStore, Mega-

Store, BigTable, HBase, that do not give up the consistency requirement and pay the availability and latency costs for it. Systems that favor low latency/high availability over consistency are Dynamo, Cassandra, Riak, PNuts [35] [64] [72] [26]. The tradeoffs involved in building distributed database systems are complex and have yet to be fully explored. Another approach is to examine the three-way tradeoff between fairness, isolation, and throughput. If the system can delay or selectively prioritize transactions throughput and consistency could be managed via this means [1]. Researchers have not formally defined fairness but a working definition of fairness is a system that does not deliberately prioritize or delay certain transactions. In a fair system, there is never an attempt to artificially add latency to a transaction in order to control the mix of transactions that are executed at a given time.

There has been recent work in trading fairness in order to obtain good throughput [37]. In-memory databases accumulate log records of database changes and write them, as a group in order to honor the durability of the ACID properties. Certain transactions cannot commit until the batch of log records exceed a certain threshold. These transactions suffer a loss of fairness for the opportunity to increase throughput.

Lazy transaction evaluation affects fairness by exploiting spatial locality when performing updates. Certain transactions with overlapping data sets are deferred to be executed in a batch fashion rather than when the data is requested [46]. This benefits throughput but at the expense of response time for certain queries.

Some kind of coordination is required to ensure distributed database consistency. Those systems guaranteeing strong isolation experience a decrease in consistency. When conflicting transactions are deferred because of coordination costs incurred under strong isolation, throughput is limited. A system that supports strong isolation must find a way to circumvent the coordination penalty. One way is to forfeit fairness whereby the database system chooses the most opportune moment to pay the cost of

coordination.

Fairness is sacrificed when a system prioritizes or delays certain transactions according to some criteria. Giving up fairness allows the system to have good performance while guaranteeing strong isolation. Fairness means that there is a best effort to quickly execute each transaction. How a system decides to pay for coordination costs results in systems with strong fairness, good throughput, but weak consistency or strong fairness, lower throughput, and strong consistency, or a system with less fairness, high throughput, and strong consistency. Again we are constrained to choose two out of three non-functional requirements.

Several distributed database systems exploit the FIT tradeoff, which is the tradeoff among fairness, isolation, and throughput requirements, in order to achieve prescribed goals. One example is G-Store, a distributed key-value store tailored for applications exhibiting spatial locality such as multi-player online games [30]. G-Store uses a key-group mechanism to avoid having to use a distributed commit protocol. Transactions belonging to a certain game are assigned to a single keygroup and all assigned transactions are executed on a single node. The cost of the grouping protocol is paid prior to transactions' executions.

Calvin is a database system that transfers the coordination cost for transactions to a preprocessing step [106]. The preprocessing step imposes a total order on transactions and processes transactions such that the serialization order is consistent with the total order of the transaction. This implies a guarantee of strong isolation. This process also eliminates deadlocks and is deterministic in that each partition is required to execute the transaction in the predetermined order. This is a form of coordination, a preprocessing step that is paid upfront. Fairness is sacrificed by having to wait for a group of transactions to accumulate and to be preprocessed.

There is applicability of the concept of fairness to the future of distributed computing, which is multi-core architecture. Contention in shared memory has shown

severe degradation of performance due to cache coherence overhead [18]. The cost of contention has the same impact as distributed coordination costs in distributed systems. This implies that the FIT analysis can be applied to multicore environments.

Another in-memory database system, Silo, locally amasses transaction log records before outputting them to disk. The reduction in sychronization comes at the expense of fairness. Certain transactions are unable to commit until the log records are flushed to disk [109]. Doppel periodically replicates high contention records, or hot records, across all cores. Commuting operations can execute on any record. The process of replicating hot records blocks the execution of transactions dependent upon those records, sacrificing fairness [77].

The analyses of the limitations to scaling distributed database systems clearly shows that, currently, some desirable feature of such systems must be sacrificed. Whatever tradeoff theory is espoused, relaxed consistency often becomes a preferred option to the development of large-scale distributed databases. Therefore, we have seen concurrency control mechanisms deployed in such databases that relax the requirement for strict consistency.

2.6 Clustering and Allocation Algorithms

There is a large body of work on the topic of clustering and many algorithms have been developed to perform the clustering task [102] [57] [114]. Clustering, or cluster analysis is often used as part of the solution to a larger problem. In addition to machine learning, clustering has been used in many disciplines in order to group data, such as pattern recognition, information retrieval, and data mining.

There are four main steps to using cluster analysis as part of a solution. Since clustering can be formulated as a multi-objective optimization problem, the first step is to develop an optimization function that evaluates the relationship between data

items, finding the best pairs. The second step is to select the clustering method. The third step is to develop a data abstraction that usefully represents the resultant clusters. Finally, a validation method is required to evaluate the quality of the clusters produced.

The cost function, an optimization function, is an objective function that generally finds the best available values from a domain of discourse. A utility function is an optimization function that returns the highest, or maximum value from a range of values. A cost function is an optimization function that returns the lowest, or minimal value from a range of values. Approaches to clustering rely on measuring the relationship between data items in terms of distance, association, or correlation and returning objects having a minimal relationship value.

Therefore, clustering algorithms generally incorporate a cost function into the evaluation process. Incorporating a cost function into the clustering algorithm makes the process a combinatorial optimization problem. Many clustering algorithms incorporate the idea of optimality implicitly. However, some algorithms explicitly define the objective function [58].

Objects for input to the clustering process can be assigned one or more data values, each data value describing some characteristic, or attribute, of the object. The dimensionality of the object space is the number of different, defined attributes. For example, we may want to collect two data items for a group of patients: type of diabetes and blood glucose level. In this instance, then, we have a two-dimensional data set. Each object is represented by a vector of values corresponding to the attributes.

The different data scales used for clustering are qualitative, otherwise called categorical, and quantitative. Groupings based upon quality place data with the same value together. Groupings based upon quantity use some form of distance measurement. Objects are thus organized into clusters based upon some measurement of similarity or distance between data values. Attributes can be discrete or continuous.

Qualitative data is discrete. Quantitative data can be discrete or continuous.

Proximity measurements, the relationship between two objects' quantitative data, can be expressed as a distance value, an association value, or a correlation value. Distance calculations quantify the degree of difference between two objects. A common distance measurement is Euclidean distance. Association coefficients are most often used with binary data and results lie within the range of 0 to 1, where 0 represents complete dissimilarity and 1 represents complete similarity. Association coefficients can also be used, however, with non-binary data. Cosine similarity, Jaccard similarity coefficient and the Jaccard variations are commonly-used association values. The correlation coefficient measures the degree of correlation between the sets of values characterizing each pair of objects. Pearson correlation coefficient is a prime example.

The distance between two objects in Euclidean space is often used as a proximity measure in cluster analysis. We map the input data into Euclidean space. The Euclidean distance in one-dimensional space for two points p and q is $d(p,q) = \sqrt{(p-q)^2} = |p-q|$.

The Euclidean distance in n-dimensional space, corresponding to a vector of n numerical attributes for each data point, is
$d(p,q) = \sqrt{(p_1-q_1)^2 + (p_2-q_2)^2 + ... + (p_n-q_n)^2}$ or $d(p,q) = \sqrt{\sum_{k=1}^{n} |p_k - q_k|^2}$.

Other forms of correspondence analysis involve binary vectors. An important similarity measure is the Jaccard index, also known as the Jaccard similarity coefficient. The Jaccard index uses binary values where an attribute, if it is present in a set, is represented by a 1, if absent, a 0 and is expressed as $J(p,q) = \frac{A_{11}}{A_{01}+A_{10}+A_{00}}$.

1. A_{11} represents the number of attributes that are 1 in p and 1 in q
2. A_{01} represents the number of attributes that are 0 in p and 1 in q
3. A_{10} represents the number of attributes that are 1 in p and 0 in q

4. A_{00} represents the number of attributes that are 0 in p and 0 in q

A variation of the Jaccard index is the Jaccard distance, which simply measures dissimilarity between sample sets by subtracting the Jaccard index from $1, J_d(P, Q) = 1 - J(P, Q)$.

Cluster properties are

1. Flat, hierarchical - The flat, or unnested, model groups data items into one set of clusters. Data items can also be grouped beginning with singleton data sets, incrementally grouping until only one large cluster is left. This is a multilevel representation where singletons are at a base level and one large cluster is at the topmost level. This is the hierarchical, or nested, representation.
2. Exclusive, overlapping, fuzzy - Data items can belong to one and only one cluster which would be exclusive. If so, than the cluster is a partition, as defined in set theory. Alternatively, data items could belong to more than one cluster and would be labeled overlapping. Third, each data item could be assigned a vector of membership ratings, each rating defining its level of affiliation with each cluster in the cluster set. This would be the fuzzy approach to cluster membership. Clusters that are overlapping or fuzzy can still be called partitions but, not in the set theoretic sense.
3. Complete, partial - In a complete clustering method, each data item must be assigned to a cluster. In a partial clustering method, not every data item must necessarily belong to a cluster.

Cluster representations are

1. Well-separated - A set of data points form a cluster if points within a cluster are all closer to each other than to any point outside of the group. Threshold values can be used to indicate correct level of closeness.
2. Nearest-neighbor - Data points in a cluster are closer to one or more points in the cluster than to any points outside of the cluster.
3. Centroid-based - Each cluster is assigned a center, or representative member. Membership is determined by the degree to which data items match the prototype. The representative member can be a centroid, which is an average of all the points assigned to a cluster up to that moment. The representative member can also be a medoid, which is an actual data point from the cluster that is most representative of that group.
4. Graph-based - A cluster is represented by a node that is the data point. Edges connect the data points, representing distances between the nodes. If the distance is below a certain threshold, then there is membership in the cluster.
5. Density-based - A cluster is defined as a dense region of data items surrounded by a sparse region of data items.
6. Similarity-based - A cluster is comprised of a set of objects that are more similar to one another than to the objects in other clusters.

Many methods, and subsequent algorithms, have been developed for clustering data [56]. Three very commonly-used clustering techniques are flat clustering analysis, hierarchical clustering, and probabilistic clustering. K-means and K-medoid are centroid-based, flat clustering techniques. Divisive and agglomerative hierarchical clustering are hierarchical clustering techniques that use either a graph-based or

geometric-based approaches to building the clusters. Density-based probabilistic classification technique use statistical methods for finding the distribution of data points.

Partitional or flat clustering methods divide the set of data items into k groups. K is an input parameter provided by the user. Data items are iteratively moved into clusters until intracluster distances have been minimized and intercluster distances have been maximized.

The advantages of such flat methods are that they are relatively scalable and handle large data sets fairly well. Such methods are also simple to understand and simple to implement. The computational complexity of such methods is $O(nki)$ where n is the number of data points, k is the number of clusters, and i is the number of iterations of the clustering process.

K-means clustering is a relatively straightforward method of clustering data items into groups [80]. The basic algorithm performs two main steps and iterates through these steps until no more changes occur. There is an assignment step where centroids are defined. Initially, the algorithm is seeded with k randomly placed centroids. The clusters are defined by finding the boundaries. For boundary identification, lines are drawn between each pair of centroids. A perpendicular line that bisects each connecting line is defined. The perpendicular lines define the boundaries of a cluster. All points within a given boundary belong to the centroid therein. The second step of the algorithm identifies a new centroid for each cluster. The new centroid is calculated using the mean or the median of the data points in the cluster so far. The process continues until the centroid values no longer change. The algorithm is as follows:

Another category of cluster analysis is hierarchical clustering. Hierarchical clustering methods are valuable in that they develop multiple levels of clusters, by building a tree, each level yielding a different number of clusters. Therefore, it becomes unnecessary to input, prior to running the clustering algorithm, the number of clusters desired. One can simply select the level in the resultant tree that corresponds to the

Algorithm 1 K-means Algorithm

Input: Set of n data items, D = $d_1, ..., d_n$, number of desired clusters, k
Output: A set of k clusters
Select a set of k points as the initial centroids.
repeat
for all d_i in D **do**
Assign each point to the nearest centroid.
Update the centroid by calculating the mean (μ) of each cluster based on the items in that cluster.
end for
until centroids no longer change

number of desired clusters.

One valuable advantage of hierarchical clustering is that one can quickly choose the level of granularity of the clusters. One can choose all the clusters from one level of the hierarchy. One can also choose significant clusters from various levels of the hierarchy. Another advantage is that many different kinds of inter-object similarity measures may be defined for clustering in this manner. One may use a number of different data values in an equation that evaluates similarity.

Disadvantages to hierarchical clustering techniques are that one must apply a termination criteria within the algorithm. Also, one may not, in most hierarchical algorithms, revisit a level in the hierarchy to redo clusters at a given level. Hierarchical clustering algorithms are greedy and the steps are irreversible.

The tree, or dendrogram, can be built in either a top-down or bottom-up fashion. The top-down clustering procedure is called divisive hierarchical clustering. The bottom-up procedure is called agglomerative hierarchical clustering. Divisive hierarchical clustering can be more accurate because it begins with a global perspective of the data. But, divisive clustering is computationally far more expensive than the agglomerative technique, where the computational cost for divisive clustering is $O(n^3)$ where n is the number of data elements. Clearly, using divisive clustering techniques on any but the smallest data sets is computationally prohibitive.

Hierarchical clustering efficiencies are $O(n^2)$ for the agglomerative approach. These efficiencies can be improved by incorporating pre-clustering techniques for multiple-phase clustering. The multiple-phase techniques continuously refines the cluster from one phase to another. Canopy clustering is an example of pre-clustering [39].

For any clustering approach, a proximity metric is established using one or more of the following approaches: Euclidean, cosine similarity, Jaccard index and its variations, or Pearson correlation. For flat clustering algorithms such as k-means or k-medoid, the proximity of two individual points is measured in order to allocate to the proper clusters.

In hierarchical clustering the distance measurement is Euclidean, cosine similarity, Jaccard index and its variations, or Pearson correlation, and is generalized from individual points to all individual points with the clusters. Such a proximity measurement is called a linkage metric. There are certain linkage metrics that have traditionally been used to join clusters together in the hierarchical methods. Classical linkage metrics used to join clusters are single-linkage, complete-linkage, average linkage, and Ward's method [16].

Single linkage measures the distance of the closest two points from each of two clusters. The single linkage metric finds the two closest data points that are in different clusters. Single-linkage metric is formulated as $min\{d(p,q) : p \in P, q \in Q\}$. When the two closest data points are found from the different clusters, the entire clusters are joined into a larger cluster for the agglomerative approach. The advantage to this type of approach is that it is very straightforward. The disadvantage is that it tends to create long chainlike clusters with some datasets. The worst-case time complexity of single-linkage metric is $O(n^2)$.

Complete linkage measures the farthest two points. The complete-linkage metric finds and calculates the distance of the two farthest data points between clusters. The algorithm then finds the smallest of these calculations and joins those clusters. This

is formulated as $max\{d(p,q) : p \in P, q \in Q\}$. The complete-linkage metric avoids the long chaining that can occur with single-linkage and is less sensitive to outliers than single-linkage techniques. The worst-case time complexity of complete-linkage metric is $O(n^2 logn)$.

Average linkage take the average distance of all pairs of points between two clusters. The group average metric calculates the average of all the pairwise distances across two clusters and picks the smallest average value, joining those clusters. The group-average metric is formulated as $\frac{1}{|P||Q|} \sum_{q \in Q} \sum_{p \in P} d(p,q)$. Average-linkage metric is a compromise between single-linkage and complete-linkage approaches. It has less sensitivity to outliers than the single-linkage approach. The time complexity of the average-linkage metric is $O(n^2 logn)$.

With Ward's method, the total within-cluster sum-of-squares is used to determine which clusters to merge. Ward's method defines the proximity between two clusters to be an increase in the squared error when the two clusters are merged. This technique is very similar to the group average method. Ward's method is the correct hierarchical analogue to the k-means approach. It is otherwise known as Ward's minimum variance method. Ward's metric is formulated as $d(P,Q) = |P - Q|^2$ or the squared Euclidean distance between points in the clusters.

Agglomerative clustering has been the dominant approach to hierarchical classification schemes [75]. The method starts with singleton clusters and successively merges pairs of clusters until all the clusters have been merged into one final cluster. This creates a hierarchy of clusters. The user can then select clusters at the level of needed granularity. The partitions can be visualized as a tree structure, called a dendrogram. Such clustering algorithms greedily merge the closest clusters. This clustering method does not need a predefined number of clusters, k, as part of the input.

The basic algorithm for hierarchical agglomerative clustering places each data

item into its own singleton group. The algorithm then iteratively merges the two closest groups until all the data has been merged into a single cluster. The algorithm results in a sequence of groupings. It is up to the user to choose the desired subset of clusters from this sequence. Groups that merge at high values relative to the merger values of their subgroups are candidates for natural clusters.

Algorithm 2 Hierarchical Agglomerative Clustering Algorithm

Input: Set of singleton clusters, $C_1, ..., C_n$
Output: A hierarchy of clusters
  repeat
    Merge the two nearest clusters.
  until until there is only one cluster left

A data abstraction is a compact description of the cluster that has been created. The cluster abstraction can be a useful and meaningful interpretation of the data found within a cluster. This description can be used for the purposes of creating a prototype, or centroid, as we have already seen. The abstracted version of the data is used as reference point in the clustering process. Data abstraction can also be used to efficiently represent the data in a cluster for further processing or for analysis by humans. The abstraction, therefore, is simply an efficient representation of the data in a cluster and can be used for different purposes [57].

The cluster representation becomes an important tool for decision-making. Data abstraction can be a simple and intuitive explanation of clusters for humans to understand. It is a form of data compression, which, as mentioned before, forms the basis for further processing. It increases the efficiency of processing. For example, a cluster of documents can be described by a representative document. When a search is performed it is done against the representative documents, not each document.

There are graphical representations of the clustering. A cluster can be represented by its centroid, a popular scheme. This representation works well for compact

clusters, but not for ones that are chainlike or irregularly-shaped. A cluster can be represented by its most distant points. Boundary points are good to offer information about the shape of the cluster. Conjunctive logical expressions can be used to represent clusters.

It is also important to assess the quality of the clusters that have been built. Even data with no natural clusterings can be grouped. Any clustering algorithm will form some groupings. Different algorithmic approaches will form different clusters. Even running the same algorithm on the same data, though with different parameters, can produce different results. Whether these groupings make any sense or not is another question entirely.

One must provide users with a degree of confidence that their clusterings make sense. A good clustering is one that achieves high intracluster similarity and low intercluster similarity. An objective function measures the quality of the clustering. Cluster quality evaluation generally takes three forms, using external evaluation, internal evaluation, or relative evaluation. An external evaluation compares the cluster structure to an existing structure in an external evaluation. An internal evaluation tries to assess whether the cluster structure is appropriate for the data. So, the comparison, in an internal evaluation, compares the cluster output with the input data set. A relative evaluation compares several cluster structures, one to the other, in order to evaluate the goodness of a particular clustering.

Another important aspect of cluster validity is the choice of the appropriate number of clusters. It is important to find techniques for choosing the appropriate number of clusters prior to running a flat clustering algorithm. It is also important to select the proper number of clusters from a hierarchy by slicing at the proper level in the tree or by selecting the optimal set of clusters from branches of the tree.

Sometimes users can select the number of clusters, k, based upon their expertise. Sometimes, however, the appropriate number, k, comes from the data itself. The

quality for the resultant clustering often relies on the value of k that is provided. Too many clusters can make the result complicated and hard to interpret and analyze. Too few clusters leads to loss of information and misleading interpretations. One attempt to properly estimate k comes from visualization of the data. If data points can be projected onto two-dimensional Euclidean space then one can get visual feedback as to the proper number of clusters. This strategy is useful for only a restricted set of applications.

Another approach is to use indices, or stopping rules, that indicate when the optimal cluster number has been reached. These indices often measure compactness of clusters and interdistance measurements among others. Research has been performed to rank the quality of the outcome of these indices [73]. It is often recommended to use several indices to estimate optimal cluster count.

Another approach is optimization of a criterion function under a mixture-model framework. Statistically, one finds the correct number of components, k, by fitting a model with some observed data and optimizing the criterion function. The Expectation-Maximization algorithm is used to estimate the model parameters for a given k. Other approaches are heuristic and based on a variety of techniques. One such technique considers the distance from a cluster centroid to clusters in its neighborhood. The k that is persistent in the largest interval of the neighborhood parameter is considered optimal [61].

Allocation algorithms such as bin-packing and knapsack problems are well-studied and well-known. The one-dimensional, bin-packing problem is a well-known allocation algorithm. It is a method to allocate a set of items $I = 1, ..., n$ each having an associated size or weight w_i. The items are dispersed amongst a set of bins, all having the same capacity c. The problem can be extended to multi-dimensional bin packing where the additional dimensions represent other salient values of the item, other than weight.

Bin packing methods and algorithms are all computationally hard. There are many optimization approaches to take in order to speed processing. One would ideally wish to use an exact method for solving a problem. An exact method will solve a problem to optimality. This is often suitable for medium-sized problems. However, when the problem space becomes unduly large, other approaches are required to come up with a reasonable answer in a reasonable amount of time.

Some exact methods for solving a problem to optimality are linear programming, branch-and-bound programming, dynamic programming, and LaGrangian relaxation methods. The most common type of application that uses linear programming is the problem of allocating limited resources among competing activities in the best possible way. It was a method developed in the research area of operational research. Linear programming cannot handle discrete variables and all functions must be linear.

Dynamic programming addresses sequential decision-making problems and other combinatorial problems. It decomposes a problem into a family of subproblems using recursion, each recursive call tackling a subproblem.Due the recursive structure of the solution, the computation time can dramatically increase with an increase in the problem size. Dynamic programming can handle discrete variables and non-linear functions.

The branch-and-bound programming model is a divide-and-conquer approach which continues to divide a problem into smaller and smaller problems to be solved. The branching corresponds to partitioning the entire set of feasible solutions into smaller and smaller subsets by fixing an integer variable's value at each iteration. The bounds phase calculates the bounds of the remaining solution spaces. The final stage discards or keeps solution spaces based upon information in their bounds. The algorithm only searches solution spaces that could contain an optimal solution.

Online environments receive data in a stream and algorithms must cope with a

constant stream of data points where there is no foreknowledge of the incoming data. The online k-means clustering algorithm keeps the clusters as close to optimal as possible as the data points stream in.

An online algorithm is meant to be able to process data coming into the program in serial fashion. Because an online algorithm suffers the disadvantage of not having a view of all of the data, it is forced to make decisions in a greedy fashion, which could force a solution into a local optimum. Online algorithms must produce results in an acceptable time bound. Recomputing entire data sets is often computationally prohibitive so approximation algorithms are often used in such a settings [8]. An approximation algorithm, unlike heuristics, guarantees a solution within a certain factor of optimum. A heuristic, however, can produce quality results, but cannot guarantee those quality results. In the area of clustering in dynamic environments, there has been some research done as to how to recreate the clusters in a computationally-acceptable way [60].

Chapter 3

Phase One - Partitioning and Allocation Framework for Atomic Web Services

Our aim is to provide efficient partitioning and replication of data. We provide only the necessary data for the anticipated workload; it is a fragmentation and allocation technique based upon the concept of minimal data sets [23] [67]. In this chapter we present a method for grouping queries by data similarity. We then define the data requirements for such groups; we call such a definition an abstract query. Data partitions are realized from the abstract queries. We present the results of an implementation using this partitioning method and show that throughput increases and latency is reduced when the database is partitioned using this method.

3.1 Introduction

There is a need to provide efficient data partitioning and allocation for services and service compositions in the context of both centralized and distributed systems [55] [28] [104] [20] [85] [4] [2]. Current web services solutions are based on full replication [69] [99] [84] [13]. However this approach involves replica updates, which can be costly [50].

We develop methods to identify data requirements of atomic web services and then groups services with similar data needs. An atomic web service, WS, is a service that has only one query associated with it. A cluster, Cl, is a grouping of WS that have similar data needs. An abstract query, Q, is created to represent the data

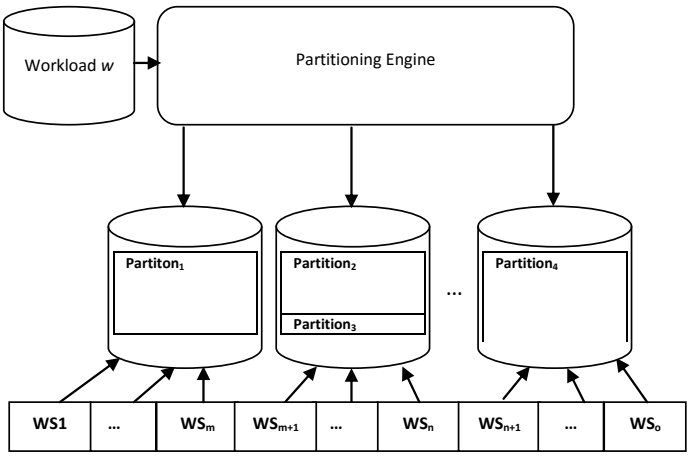

Figure 3.1: Framework for Phase I - Partitioning and Allocation System

needs of such a grouping. The projection statement, A, of Q is built from attribute information of the participating queries. The selection statement, C, of Q is built from predicates of the participating queries. Q contains the information necessary to form a micropartition. We assume that all services and nodes are in a central location, as in a cloud. Figure 3.1 shows our framework.

We show that our abstract query definition is complete and minimal. Our empirical results show that our approach improves data access efficiency over standard partitioning of data. Our method shows an increase in throughput of 110% over standard range partitioning and a decrease in latency of 38%.

In Section 3.2 we present the definitions and procedures for creating the abstract query, Q. In Section 3.3, we walk through an example of abstract query development. Section 3.4 develops the clustering cost function, p, used to group queries in a workload. In Section 3.5 we present the clustering algorithm and the allocation algorithm, respectively. In Section 3.6 we describe our implementation and show results. In Section 3.7 we further develop the clustering cost function.

3.2 Definitions

In this section we define atomic web service, WS and abstract query, Q. We define the projection statement, A, for an abstract query and present the algorithm that constructs the projection statement. We define the selection statement, C, for an abstract query and present the algorithms that construct the selection statement. We then show that the abstract query definition is complete and minimal.

Definition 1: Atomic Web Service WS

An atomic web service, WS, is a sequence of read and write operations on data items and a commit or an abort. We represent each WS as a relational algebra query, Q, of the form $Q = \pi_A \sigma_C(D)$, where D is a universal relation, A is a subset of attributes of D, and C is a conjunction of boolean expressions of the form $(a_1$ op $a_2)$, where $a_i (i = 1, 2)$ are either attribute names or constants and *op* is a relational operator $=, <$, or $\leq$.

Definition 2: Workload w

The workload, w, is a pair $(WS_w, \leq_w)$ where WS_w is a set of atomic web services $\{WS_1, ..., WS_n\}$ and $\leq_w$ is the partial order of WS_j where $(j = 1, ..., n)$.

We use the concept of abstract query to establish data partitioning. Intuitively, an abstract query represents a set of atomic services with similar data needs. A data partition, i.e., the answer generated for an abstract query, can satisfy the data needs of all the services represented by the abstract query.

Definition 3: Abstract Query Q

Let Cl be a cluster of atomic web services denoted as $Cl = \{Q_1, ..., Q_n\}$ where each query Q_j is of the form $Q_j = \pi_{A_j}\sigma_{C_j}(D)$, for $1 \leq j \leq n$ where D is a universal relation. We say that Q, denoted as $Q = \pi_A\sigma_C(D)$, is the abstract query representing Cl if the following hold:

For any database D

1. (Containment) $Q_j(D) \subseteq Q(D)$ for all $1 \leq j \leq n$ and
2. (Minimality) $\nexists\, Q^{'}(D) \subset Q(D)$ and $Q_j(D) \subseteq Q^{'}(D)$ for all $1 \leq j \leq n$

The projection and selection statements for abstract query Q are built from a set of query definitions in a cluster. That is, the properties of A are

1. $CA_j \cup A_j \subseteq A$ for all $j = 1, ..., n$
2. There is no $A^{'}$ such that $CA_j \cup A_j \subseteq A^{'}$ for all $j = 1, ..., n$ and $A^{'} \subseteq A$ and $A \neq A^{'}$

and the properties of C are

1. $C_j \Rightarrow C$ for all $j = 1, ..., n$
2. There is no $C^{'}$ such that $C_j \Rightarrow C^{'}$ for all $j = 1, ..., n$ and $C^{'} \Rightarrow C$ and $C \not\Rightarrow C^{'}$

Intuitively, condition 1 for A ensures that any attribute that participates in the answer for A_j for all (j=1,...,n) will also be in the answer for A. Condition 2 ensures that no projection statement $A^{'}$ exists that is more restrictive than A, and the abstract query $Q^{'}$ conditioned on $A^{'}$ would contain the answers for A_j for all (j=1,...,n) but would not contain some of the answer returned for Q.

Intuitively, condition 1 for C ensures that any tuple that participates in the answer for C_j for all (j=1,...,n) will also be in the answer for C. Condition 2 ensures that no selection condition $C^{'}$ exists that is more restrictive than C, and the abstract query $Q^{'}$ conditioned on $C^{'}$ would contain the answers for C_j for all (j=1,...,n) but

would not contain some of the answer returned for Q.

Projection Statement Construction

With the modified C_j assigned to Cl, we now create an abstract query, Q, for each Cl. We form the selection statement, C, for each Q by first removing each C_i in Cl where $C_i \Rightarrow C_j$. We the join all remaining C_j via disjunction. The individual conjunctive clauses in C are reduced by removing redundant subgoals. The final C for Q represents a complete and minimal requirement of the tuples needed for all queries associated with cluster Cl.

To build the projection statement for Q, we take the union of all attributes in the projection and selection statements for each query belonging to a cluster.

Let $Cl = \{Q_1, ..., Q_n\}$.

We create the attribute set, A, of the abstract query $Q = \pi_A \sigma_C(D)$, as

$A = \{A_1 \cup ... \cup A_n \cup CA_1 \cup ... \cup CA_n\}$, where CA_j is the set of attribute names that appear in C_j for all $1 \leq j \leq n$.

Algorithm 3 computes the projection attributes, A, for the abstract query, given a cluster, Cl.

Algorithm 3 BuildProjectionStatement

Input: $Cl = Q_1, ..., Q_n$
Output: A
  $A = \{\emptyset\}$
  for all Q_j in Cl **do**
    $A \leftarrow A_j \cup A$
    $A \leftarrow CA_j \cup A$
  end forreturn$A$

Selection Statement Construction

In our definition for an atomic web service, WS, we defined the projection statement as a list of attributes, A, and the selection statement as a conjunctive clause, C. The data structure representation is shown in Figure 3.2. The workload w consists of the set of atomic web services, WS_w and their partial order information.

A_w	
A_1	$A_1,\ldots,A_m$
...	...
A_n	...

C_w	
C_1	$p_1 \wedge \ldots \wedge p_n$
...	...
C_n	...

Figure 3.2: Initial Data Structures Representing WS_w

Prior to the clustering process, we prepare each C_j, from WS_w. We first restate all predicates, p_i, within the original C_j by converting all operators to $=, <, \leq$ (i.e., $A_i > c_i$ would be converted to $c_i < A_i$). We then apply transitive closure to each C_j. Selection of new operators for additional predicates is shown in Figure **??**. To apply transitive closure to each C_j, we use WS_w as input to Algorithm 5 (modified Warshall's algorithm). By the algorithm, we generate the transitive closure for each selection statement in WS_w.

We build the selection statement, C for Q of each Cl. Prior to clustering process, we prepare each C_j, from each WS_j, $1 \leq j \leq n$, in workload w. We first restate all predicates within the original C_j by converting all operators to $=, <, \leq$ (i.e., a predicate, $A_i > c_i$ would be converted to $c_i < A_i$). We then find transitive closure on each C_j.

With the modified C_j assigned to Cl, we now create abstract query, Q, for each

Cl. We form the selection statement, C, for each Q by first removing each C_i in Cl where $C_i \Rightarrow C_j$. We the join all remaining C_j via disjunction. The individual conjunctive clauses in C are reduced by removing redundant subgoals. The final C for Q represents a complete and minimal requirement of the tuples needed for all queries associated with cluster Cl.

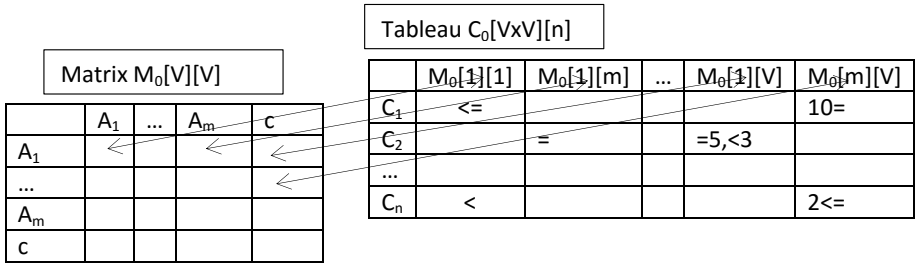

Figure 3.3: Data Structures Used to Build C

The tableau built in Algorithm 4 is used to build the final C for an abstract query Q. Data structure M_0 identifies the attributes used in a predicate or the single attribute used and a constant. Data structure C_0 identifies the relational operator used for a predicate in a given conditional statement and an associated constant if warranted. We use the data structures in Figure 3.3 in Algorithms 5 and 4 to build the abstract query Q for each cluster Cl. Algorithm 5 uses M_0 as the data structure to apply transitive closure to C_j and adds the additional predicate info to C_j in tableau C_0.

Algorithm 4 BuildC_0

Input: M_0 - table of reconstructed condition statements
Output: C_0 - tableau to build select statement for abstract query Q

```
Construct a temporary record C_k from M_0
Addit ← 1
for all C_i in C_0 do
  if C_i ⟹ C_k then
    Addit ← 0
    Break
  else if C_k ⟹ C_i then
    Delete C_i from C_0
  end if
end for
if AddIt then
  Add C_k to C_0
end if
```

Algorithm 5 TransitiveClosure

Input: $Cl = \{Q_1, ..., Q_n\}$
Output: C_0 - table of reconstructed condition statements

```
for all C_j in Cl do
  Initialize M_0
  for all operand in C_j do
    Insert row and column to M_0 for unique operands
    M_0[k][l] ← <, ≤, or =
    if operator == "=" then
      Add operator to M_0[l][k]
    end if
  end for
  n ← number of columns in M_0 - 1
for all i = 1; i <= n; i++ do
for all j = 1; j <= n; j++ do
for all k = 1; k <= n; k++ do
  if M_0[i][j] == " ≤ " || M_0[i][k] == " ≤ " then
    M_0[i][k] ⇐ " ≤ "
  else if M_0[i][j] == " < " || M_0[i][k] == " < " then
    M_0[i][k] ⇐ " < "
  else
    M_0[i][k] ⇐ " = "
  end if end for  end for  end for
  BuildC_0(M_0)
end for
BuildC(C_0)
```

To build C, from tableau C_0, we remove all redundant subgoals from each row, constructing a conjunctive clause from each row. The conjunctive clauses are then united using logical disjunction to form the final C.

In this section, we state and prove the properties of our calculation.

Theorem 1: Given a query cluster $Cl = \{Q_1, ..., Q_n\}$, Algorithms 3 and 4 compute the projection and selection statements, A and C respectively, for the corresponding abstract query, Q.

Proof Sketch:

Let Q_j be $\pi_{A_j}\sigma_{C_j}(D)$, where A_j is the extended set of attributes of Q_j, s.t. every attribute in A_j is either in the original set of projected attributes or in C_j.

Because A is the union of all the projection attributes of Q_j, for all j=1,...,n, every $A_j \subseteq A$ and there is no attribute in A that is not represented in at least one A_j in the set of queries belonging to Cl.

Second, we need to show that every tuple, returned by Q_j must also be returned by Q. Note, that for every tuple t that is returned by Q_j, t must satisfy C_j; but then, because $C_j \Rightarrow C$, t must also satisfy Q and, therefore, must be in the answer set of Q.

Third, we need to show that there is no $Q^{'}$ such that

1. $Q_j \subseteq Q^{'}$ where j=1,...,n and
2. $Q^{'} \subset Q$ and
3. $Q^{'} \not\supseteq Q$

We show this by contradiction. Assume that there is a tuple t such that t satisfies C (i.e., t is in the answer to Q) but t does not satisfy C', therefore t is not in the answer to Q'.

We know that $C_j \implies C$, because any t that satisfies C must also satisfy C_j. Let

us assume there is a C' s.t. $C_j \implies C'$, that is, any tuple t that satisfies C' must also satisfy C_j. Where $C' \implies C$ but $C \not\Longleftrightarrow C'$, this means that C is more general than C'. By contradiction, we state that there exists a tuple t that satisfies C but does not satisfy C_j or C'. Then, there is a condition (preposition), p_k, in C' such that p_k is not true for t. But then, p_k was removed from C (but remained in C') due to the generalization of conditions by Algorithm 4. Initially, both C' and C are constructed as a disjunction of selection conditions. But then, p_k was generalized for the construction of C in two possible places:

1. By Algorithm 5, where the transitive closure of each C_j was generated. However, Algorithm 5 does not remove any selection condition, so both C' and C must have p_k. This is a contradiction to the original statement that p_k is in C' but not in C.
2. The second place where p_k could have been removed is in Algorithm 4 to build a combined condition, however, in this step, we drop C_j when we have analyzed that another C_k in WS provides all the tuples required by C_j. This will happen for both C' and C. This is also a contradiction to the original statement that p_k is in C' but not in C.

□

3.3 Abstract Query Example

Building the Projection Statement Example

The projection statement, A, for the abstract query, Q, is built by the taking all unique attributes found in the projection and selection statements of each Q_i in Cl. The projection statement for Q is the union of all modified projection statements of the queries in Cl. Q_1 and Q_2 are example queries that have been extracted from the well-known TPC-C benchmark. See Figure 3.4 for the TPC-C database layout.

Q_1: πcustomer.c_discount, customer.c_last, customer.c_credit, warehouse.w_tax

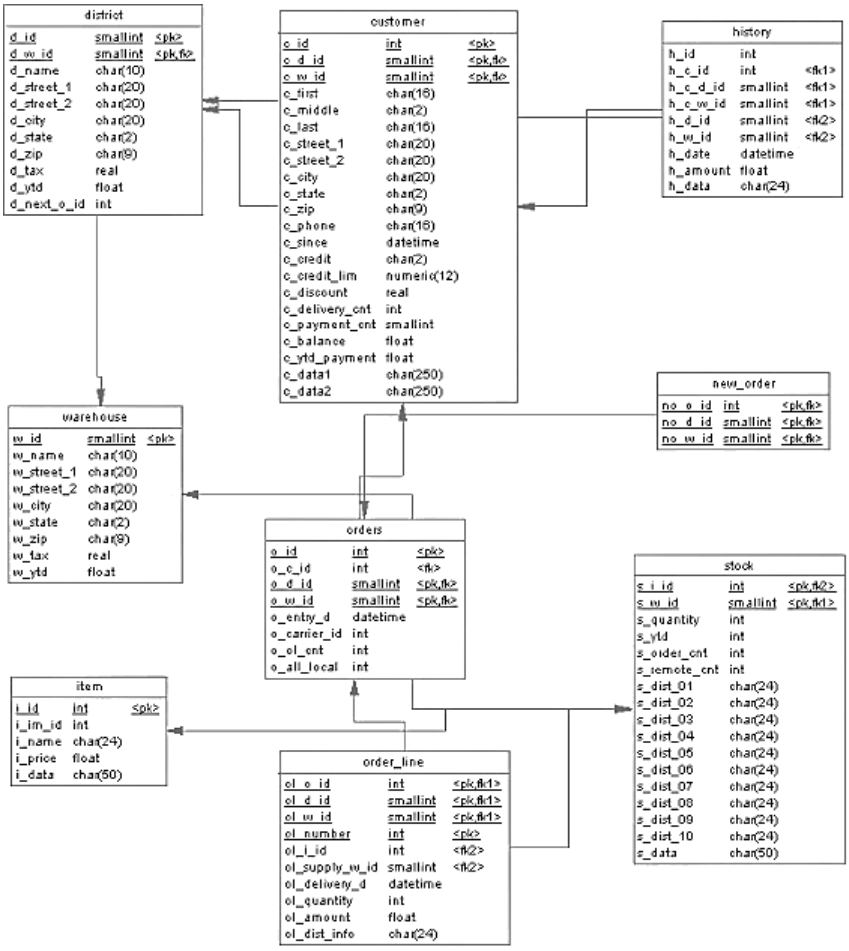

Figure 3.4: TPC-C Schema (source: [49])

(σcustomer.c_w_id $<$ 5 (σ warehouse.w_id = customer.c_w_id (customer $\bowtie$ warehouse)))

Q_2: πc_data1 (σc_w_id $<$ 5 (customer))

Attribute set $A = \{A_1 \cup ... \cup A_n \cup CA_1 \cup ... \cup CA_n\}$, where CA is the set of attribute names.

A_1: c_discount, c_last, c_credit, w_tax, w_id, c_w_id

A_2: c_data1, c_w_id

A: c_discount, c_last, c_credit, w_tax, w_id, c_w_id, c_data1

Building the Selection Statement

Create the selection statement, C, for the abstract query, Q. We begin with the original conditions from each query in Cl.

C_1: w_id $< 5 \wedge$ c_w_id = w_id

C_2: c_w_id < 5

Figure 3.5 shows the intuition behind our transitive closure. We apply transitive closure to each selection statement. In this example, we show the application of transitive closure for C_1. We do this by using Warshall's Algorithm to complete a reachability graph. For C_1, we add an edge to the predicate with the equality operator, indicating that we are adding the predicate, w_id = c_w_id. For C_1, an additional directed edge has been added between nodes cwId and 5, indicating that we are adding a new predicate, c_w_id < 5, to the modified selection statement.

C_1: w_id $< 5 \wedge$ c_w_id = w_id $\wedge$ w_id = c_w_id $\wedge$ c_w_id < 5

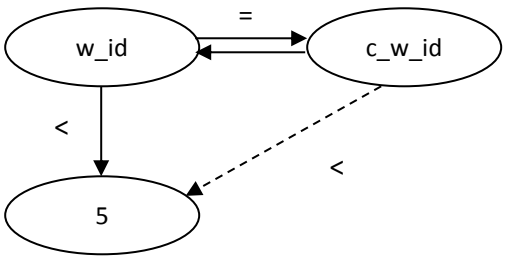

Figure 3.5: Transitive Closure on Conditions in C_1

The select statements to which transitive closure has been applied now include any new predicates. We now remove predicates with user-defined variables (? in SQL

statement predicates would be an example of this) for the final, modified selection statements. We reduce C by dropping clauses where there is containment. Since, $C_1 \not\rightarrow C_2$ nor $C_2 \not\rightarrow C_1$, neither clause is dropped. Second, we eliminate redundant subgoals. With C_1 and C_2 modified, we create the final selection statement C by combining C_1 and C_2 via a logical $\vee$.

C_1: w_id $< 5 \wedge$ c_w_id = w_id $\wedge$ w_id = c_w_id $\wedge$ c_w_id < 5

C_2: c_w_id < 5 C: w_id $< 5 \wedge$ c_w_id = w_id $\vee$ c_w_id $= 5$

Next, we build the selection statement for Q. We begin by restructuring the selection statement of each query in Cl. Every Q_i in Cl will have a selection statement of the form $\rho_1 \wedge, ..., \wedge \rho_n$ where the operands in the predicates are either an attribute of D or a constant and operators in ρ are $=, <$, or $\leq$.
The selection statement, C, for the abstract query Q, is computed as follows:
Let $Q = \pi_A \sigma_C(D)$ denote the abstract query for $Cl = \{Q_1, ..., Q_n\}$.

The selection statement, denoted as C, of Q, is defined as
$C = \{C_1 \vee ... \vee C_n\}$ for all $1 \leq j \leq n$,
To satisfy the intuition for the selection statement, we compute C as follows:

The abstract query, Q, minimally covers the data requirements for both Q_1 and Q_2.

Q_1: π customer.c_discount, customer.c_last, customer.c_credit, warehouse.w_tax, warehouse.w_id,
customer.c_w_id (σ customer.c_w_id ≤ 5 (σwarehouse.w_id = customer.c_w_id (σc_w_id < 5 (customer $\bowtie$ warehouse))))

Q_2: πc_data1, c_w_id, c_d_id (σ c_w_id ≤ 5 (customer))

Q: πcustomer.c_discount, customer.c_last, customer.c_credit, warehouse.w_tax, warehouse.w_id, customer.c_w_id, customer.c_data1, customer.c_w_id, customer.c_d_id (σ w_id $<$ 5 (σ c_w_id = w_id (σ c_w_id $<$ 5 (σ c_w_id $\leq$ 5 (customer $\bowtie$ warehouse)))))

Finally, we reduce C by merging clauses where there is containment and by eliminating redundant subgoals.

Q: πcustomer.c_discount, customer.c_last, customer.c_credit, warehouse.w_tax, warehouse.w_id, customer.c_w_id, customer.c_data1, customer.c_w_id, customer.c_d_id (σ c_w_id $\leq$ 5 (customer $\bowtie$ warehouse))

3.4 Clustering Cost Function

We present our method to cluster queries based on similar data needs. The cost function, for our clustering, includes proximity measurements, v (attribute similarity) and h (selection statement similarity); the values of h and v together determine the overall proximity value, p. The value p is defined later in this section.

The measurement v is the vertical partitioning component of the clustering; h is the measure for horizontal partitioning. The first measurement, v, measures the similarity between the attributes of two abstract queries, Q_i and Q_j. The second calculation, h, measures the similarity of the condition ranges of Q_i and Q_j. We select the clusters with the largest p value, to merge.

Similarity of Query Attributes Expression

The first measurement, v, uses Jaccard coefficient to measure the similarity between the attributes in Q_i and Q_j. We merge the two abstract queries having the most similarity (value closest to 1.0) with respect to their attributes.

For abstract queries Q_i and Q_j, v measures the similarity between the respective attribute sets, A_i and A_j.

$v = J(Q_i, Q_j) = |A_i \cap A_j| / |A_i \cup A_j|$

For example, given the following queries Q_1 and Q_2

Q_1: πc_discount, c_last, c_credit, w_tax, w_id, c_w_id σ w_id $< 5 \wedge$ c_w_id = wId $\wedge$ c_w_id < 5

Q_2: πc_data, c_w_id, c_d_id σ c_w_id ≤ 5

and their respective attribute sets

A_1: cDiscount, cLast, cCredit, wTax, wId, cwId

A_2: cData, cwId

$v = J(Q_i, Q_j) = \frac{1}{1+5+1} = \frac{1}{7} = 0.14$

Similarity of Query Condition Ranges Expression

We calculate the second similarity measurement, h by evaluating the selection condition ranges for abstract queries Q_i and Q_j. This is a measurement of tuples returned, and, therefore, of horizontal partitioning.

We have developed a measurement based on the database schema and corresponding metadata and not on the actual database instances. For our initial analysis we apply the following simplifying conditions: (1) All attributes used in conditions have attribute domain restrictions that are defined in the database schema and (2) conditions take the form of conjunctions of (a_1 *op* a_2), (a_1 *op* c), or (a_1 *op* ?) where a_1 and a_2 are attributes in the universal database, D. Operator *op* is a standard comparison operator, =, < or, <=. Constant c is a constant value. (3) Data is evenly distributed across the range of schema values.

For abstract queries Q_i and Q_j, h measures the similarity between the respective condition ranges, R_i and R_j.

$h = |R_i \cap R_j| / |R_i \cup R_j|$

For example, given the following abstract queries

Q_1: πc_discount, c_last, c_credit, w_tax, w_id, c_w_id σw_id $< 5 \wedge$ c_w_id = w_id $\wedge$ c_w_id < 5

Q_2: πc_data1, c_w_id, c_d_id σc_w_id ≤ 5

and their respective ranges, R_i and R_j, where R_i and R_j must contain integer values within the domain of cwId, which is 1-10.

R_i: $1, 2, 3, 4$

R_j: $1, 2, 3, 4, 5$

$h = \frac{4}{5} = 0.80$

Proximity Measure

Our final proximity measure incorporates both v and h.

Definition 4: Proximity measure, p, calculates the similarity value of two abstract queries, Q_i and Q_j.

$p = w_v * v + w_h * h$

where w_v and w_h are weights assigned to v and h. Initially, w_v and w_h are both 0.5.

Consider our running example; the final proximity value is $p = 0.5*0.14+0.5*0.80 = 0.47$

3.5 Clustering

Queries for available services are exposed in the service inventory or can be exposed in a workload trace. We also have a workload, w_{WS}, and the first task is to gather

all the associated exposed queries from the service inventory. The abstract query for workload query Q_i is trivially created by replacing attributes in the projection statement for Q_i with the union of A_i and CA_i and also applying transitive closure to the selection statement of Q_i and removing any user-defined conditions.

We cluster WS by similarity measure, p, using agglomerative hierarchical clustering technique. Generally, a dendrogram is a good representation of hierarchical clustering. The leaves of our representation would be the abstract query representation of each workload query. This is level 0 in the tree. Each of these level 0 queries will have an assigned similarity, p, value of 1.0.

Proximity measurement, $p = (w_v * v + w_h * h)$, is calculated and stored, in a proximity matrix, for each query pair, WS_iWS_j, for workload w_{WS}. We use the average linkage measurement to calculate p for all clusters in the hierarchy. We use a standard clustering algorithm to build the tree.

Algorithm 6 HierarchicalClustering Algorithm

Input: $Q_{w_{WS}} = \{Q_1, ..., Q_n\}$ - set of all queries from the workload
Output: Proximity matrix
  Compute initial proximity matrix from $Q_{w_{WS}}$;
  Number of Clusters $\Leftarrow n$
  while Number of Clusters > 1 **do**
    Merge two "closest" clusters by finding the largest p;
    Update proximity matrix;
  end while

Each level of the tree represents a newly-formed cluster, except for the first level which which represents all the singleton clusters. We now label each level of the hierarchy with the proximity value, p, for all the queries that participate in that cluster. The proximity values, from level 0 to the top-level, will be monotonically-decreasing from a starting value of 1.0.

We use the largest gap method for selecting the level of the tree at which to select clusters. Large gaps between levels often indicate natural clusterings, points where

adding one more cluster decreases the quality of the clustering significantly. We cut the tree at the gap where the difference between the similarity values between levels is the largest.

Once we have selected the clusters we calculate the size of the micropartition that would be generated from a cluster's abstract query, Q. We apply the classical Worst Fit Decreasing (WFD) algorithm. The WFD algorithm is used to order the micropartitions by decreasing size. The algorithm allocates the micropartitions across k nodes beginning with the first micropartition entry, which is the largest. Then, the next largest micropartition is fitted to the nodes with the most non-allocated space until all micropartitions have been allocated. The WFD algorithm is a heuristic variant of the general bin-packing algorithm and offers good performance.

Algorithm 7 WFD Algorithm

Input: - $Cl = \{Cl_l, ..., Cl_n\}$, cluster (includes micropartition size), $k = \{k_1, ..., k_n\}$ - nodes
Output: Updated k with cluster assignments
  Sort Cl in decreasing order of micropartition size;
  for all Cl_i in Cl **do**
    for all k_j in k **do**
      Assign to node k_j if $|k_j| < k_i$ for all $i = 1, ..., n$ where $i \neq j$
    end for
  end for

3.6 Implementation

In our implementation, we use the OLTP-Bench benchmarking system [27]. OLTP-Bench is an extensible, universal benchmarking infrastructure that centralizes several benchmarking systems. OLTP-Bench gives developers access to real and synthetic databases and a variety of workloads. The testbed is oriented toward OLTP and web-oriented workloads, making available several statistics.

1. Throughput: Average throughput sustained for a period of time.

2. Latency: Latency in microseconds while running at max-throughput.

3. Warmup Time: Elapsed time going from a cold state to maximum sustained throughput.

OLTP Benchmark Experiments

We selected, from the OLTP-Bench infrastructure, the TPC-C implementation (See Figure 3.4 for TPC-C database layout). TPC-C is an OLTP workload with a mixture of read and write intensive transactions that simulate an order-processing environment [89]. We modified the OLTP-Bench TPC-C benchmarking system, so that, instead of running 32 queries spread amongst five transactions, we remodeled the transactions so that we had fifteen transactions, each running 1 query.

Our first step was to generate the hierarchy of clusters, joining queries based on their proximity, p. We used fifteen query definitions from the workload transactions as input. We prepared the queries for input according to the process outlined in the abstract query section. We removed predicates in the selection statement that included user-defined variables. We extended the projection statement by including any attributes found in predicates in the selection statement that were not already in the selection statement. We applied transitive closure to the predicates in the selection statements. The prepared queries are listed in Table 3.1. A representation of the resulting clustering is in Figure 3.6. We see, in Figure 3.6, for example, that queries number 6 and 14 represented in Table 3.1, have exactly the same attributes and conditions and form the first cluster, 16, with full similarity and a proximity measurement of 1.0. We see that cluster 19 is formed from queries 4,9,and 10, those queries having an average similarity value of 0.81.

We selected our clusters using the largest gap method explained in Section 3.5 and chose clusters (28, 27) and (25, 24, 22, 21, 20). We then created a TPC-C database which contained information for ten warehouses, each warehouse supplying ten districts. We partitioned the TPC-C database in four different ways. The first

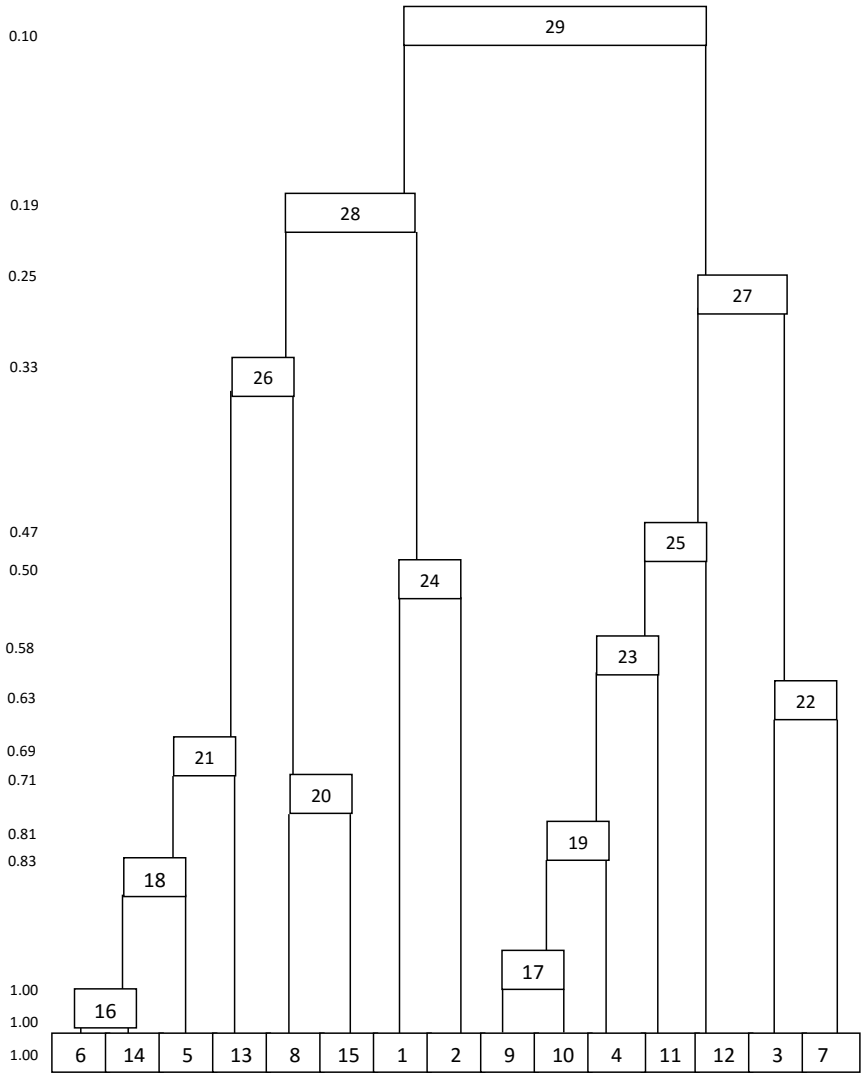

Figure 3.6: Dendrogram of Clusters Built from Input Queries

Table 3.1: Input Queries

Query No	Query Name	Input Query	Abstract Query, Q
1	Delivery1	SELECT no_o_id FROM new_order WHERE no_d_id < 3;	A_1 – no_o_id, no_d_id C_1 - no_d_id < 3
2	Delivery2	DELETE FROM new_order WHERE no_d_id > 2;	A_2 – no_d_id C_2 - 2 < no_d_id
3	Delivery3	SELECT o_c_id FROM oorder WHERE o_d_id = 2;	A_3 – o_c_id, o_d_id C_3 - o_d_id = 2 and 2 = o_d_id
4	NewOrder1	SELECT c_discount, c_last, c_credit, w_tax FROM customer, warehouse WHERE c_d_id < 4;	A_4 – c_discount, c_last, c_credit, w_tax, c_d_id C_4 - c_d_id < 4
5	NewOrder2	SELECT d_next_o_id, d_tax FROM district WHERE d_id <= 2 FOR UPDATE;	A_5 – d_next_o_id, d_tax, d_id C_5 - d_id <= 2
6	NewOrder4	UPDATE district SET d_next_o_id = d_next_o_id + 1 WHERE d_id = 2;	A_6 – d_next_o_id, d_id C_6 - d_id = 2, 2 = d_id
7	OrderStatus1	SELECT o_id, o_carrier_id, o_entry_d FROM oorder WHERE AND o_d_id < 5;	A_7 – o_id, o_carrier_id, o_entry_d, o_d_id C_7 - o_d_id < 5
8	OrderStatus2	SELECT ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_delivery_d FROM order_line WHERE ol_w_id = 2;	A_8 – ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_delivery_d, ol_w_id C_8 - ol_w_id =2 and 2 = ol_w_id
9	OrderStatus3	SELECT c_first, c_middle, c_last, c_street_1, c_street_2, c_city, c_state, c_zip, c_phone, c_credit, c_credit_lim, c_discount, c_balance, c_ytd_payment, c_payment_cnt, c_since FROM customer WHERE c_d_id < 3;	A_9 – c_first, c_middle, c_last, c_street_1, c_street_2, c_city, c_state, c_zip, c_phone, c_credit, c_credit_lim, c_discount, c_balance, c_ytd_payment, c_payment_cnt, c_since, c_d_id C_9 - c_d_id < 3

method is standard range-partitioning method on the entire database. The second method is standard range-partitioning on the topmost cluster in the hierarchy (only that data used by the queries).

The third method partitions by realizing two micropartitions, one from the abstract query description for cluster 28 and one from the abstract query description for cluster 27. The fourth, and final, method, creates five micropartitions from clusters 25, 24, 22, 21, and 20. These micropartitions are allocated using the classical WFD algorithm depicted in Section 3.5.

1. Standard range-partitioning of the entire database is a popular approach used by database managers. We created two partitions, one placed on each node. We

Table 3.2: Input Queries

Query No	Query Name	Input Query	Abstract Query, Q
10	OrderStatus4	SELECT c_first, c_middle, c_id, c_street_1, c_street_2, c_city, c_state, c_zip, c_phone, c_credit, c_credit_lim, c_discount, c_balance, c_ytd_payment, c_payment_cnt, c_since FROM customer WHERE c_d_id <= 2	A_{10} – c_first, c_middle, c_id, c_street_1, c_street_2, c_city, c_state, c_zip, c_phone, c_credit, c_credit_lim, c_discount, c_balance, c_ytd_payment, c_payment_cnt, c_since, c_d_id C_{10} - c_d_id <= 2
11	Payment1	UPDATE warehouse SET w_ytd = w_ytd + ?;	A_{11} – w_ytd C_{11} -
12	Payment2	SELECT w_street_1, w_street_2, w_city, w_state, w_zip, w_name FROM warehouse;	A_{12} – w_street_1, w_street_2, w_city, w_state, w_zip, w_name C_{12} -
13	Payment3	UPDATE district SET d_ytd = d_ytd + ? WHERE d_id <=3	A_{13} – d_ytd C_{13} – d_id <= 3
14	StockLevel1	SELECT d_next_o_id FROM district WHERE d_id = 2	A_{14} – d_next_o_id, d_id C_{14} – d_id = 2 and 2 = d_id
15	StockLevel2	SELECT COUNT(DISTINCT(s_i_id)) AS stock_count FROM order_line, stock WHERE ol_w_id = 2 AND s_w_id = 2 AND s_i_id = ol_i_id;	A_{15} – s_i_id, ol_w_id, ol_i_id C_{15} – ol_w_id = 2 and 2 = ol_w_id and s_w_id = 2 and 2 = s_w_id and s_i_id = ol_i_id and ol_i_id = s_i_id

placed information related to warehouses 1 through 5 in partition one and the information for warehouses 6 through 10 into partition two.

2. We selected the topmost cluster, (29), in the generated hierarchy. Using range partitioning on the one cluster, we created two partitions.

3. We sliced the hierarchy at $k = 2$ clusters (28 and 27) [1] We create a micropartition for each cluster, placing each micropartition on a node.

4. We sliced the hierarchy at $k = 5$ clusters (25, 24, 22, 21, 20) [2]. We create mi-

[1] See Table 3.1 and Figure 3.6 for corresponding queries and clusters.

28-NewOrder4,StockLevel1,NewOrder2,
Payment3,OrderStatus2,StockLevel2,
Delivery1,Delivery2

27-OrderStatus3,OrderStatus4,NewOrder1,
Payment1,Payment2,Delivery3,
OrderStatus1

[2] See Table 3.1 and Figure 3.6 for corresponding queries and clusters

cropartitions for each of the five clusters and allocate the micropartitions to the two nodes in a balanced way.

Empirical Results

Ten workload executions were performed against each of the four different methods of partitioning and latency and throughput information was captured every 5 seconds for the 60 second runs. Each data point on the graphs in Figures 3.7 and 3.8 represent an average of each of the twelve reported values for a 60 second run. There is considerable improvement in both latency and throughput performance when using the new clustering method (See Figures 3.7 and 3.8).

We see, in Figure 3.7, that the partitioning design based on the five clusters gives a lower latency, averaging 80 ms, then the standard two-partition design, averaging 120 ms. We see, in Figure 3.8, that the partitioning design based on the five clusters gives a higher throughput, averaging 100 req/sec, then the standard two-partition design, averaging 25 req/sec. The latency for queries running against the five-cluster micropartition is low and query throughput. Users receive a quick and similar response time for every execution.

However, the cost of generating the clusters is high. Also, given a workload with very similar queries and a higher number of selected clusters, there may be higher redundancy. Of course, redundancy is not a problem if the queries are read-only.

The averages of each of the sets of ten workload executions are listed in Table

---

25-OrderStatus3,OrderStatus4,NewOrder1,
Payment1,Payment2

24-Delivery1,Delivery2

22-Delivery3,OrderStatus1

21-NewOrder4,StockLevel1,NewOrder2,
Payment3

20-OrderStatus2,StockLevel2

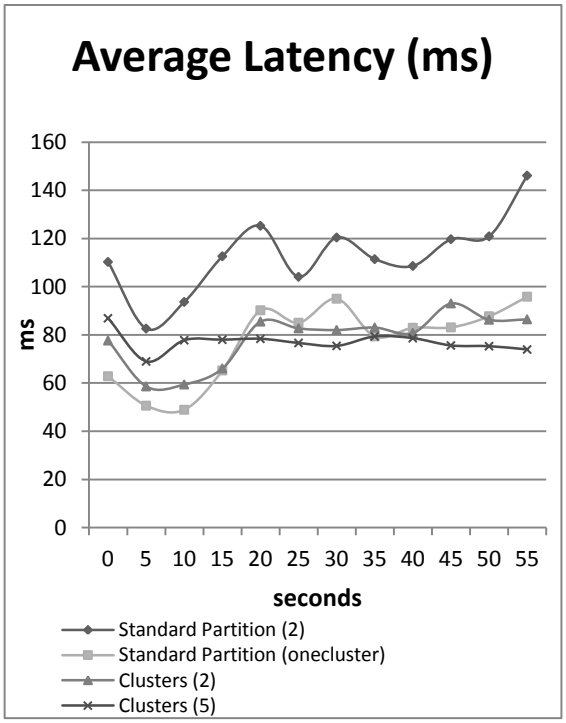

Figure 3.7: Average Latency

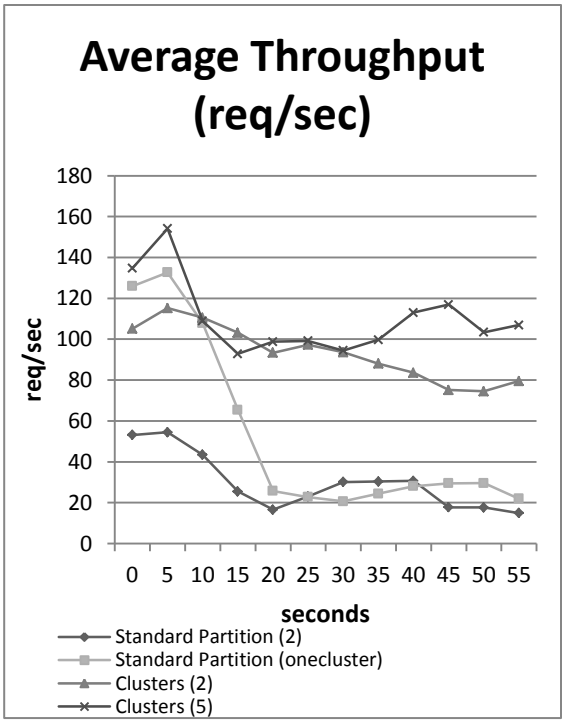

Figure 3.8: Average Throughput

3.3. The table shows the results of our experiment. We see an increase of 76% in throughput for a partitioning design using two clusters over standard partitioning using the same subset of data. We see an increase in throughput of 110% for a five cluster design. The average latency for two cluster and five cluster improved over standard partitioning method by 22% and 38% respectively.

Standard range partitioning methods balance processing well when the database is partitioned to match the access requirements of the workload. In other words, if the partitioning index matches the search conditions for the queries then access will be efficient. For example, there was balanced access for the TPC-C workload when executed against the partitioning using standard range partitioning techniques. See Graphs 3.9, 3.10, which depict that query processing is performed in a balanced way against the partitions. In other words, one partition does not have higher processing demands placed upon it than the other partition.

Our partitioning method has overall better performance for the TPC-C work-

Table 3.3: Summary Metrics

	Throughput		Latency	
Method	Average (req/sec)	Increase	Average (ms)	Decrease
Standard Partitioning 1	29.78		113.15	
Standard Partitioning 2	52.86		105.19	
Clusters (2)	93.25	+76%	82.23	-22%
Clusters (5)	110.26	+109%	65.64	-38%

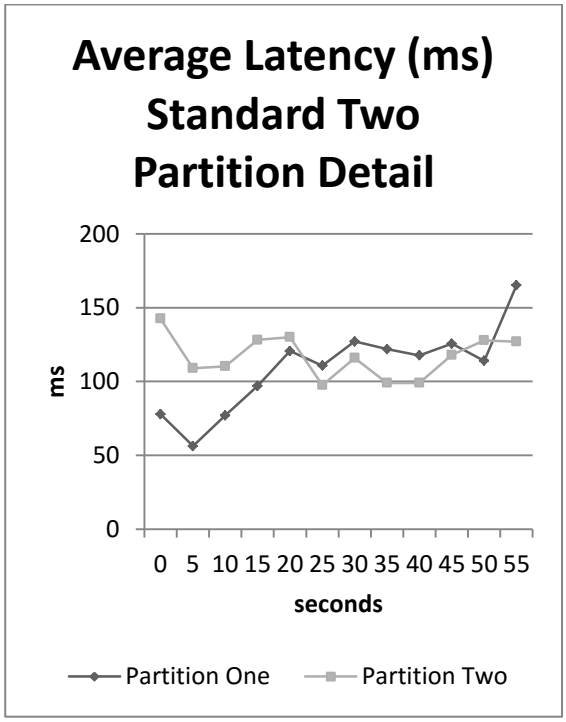

Figure 3.9: Latency Two Partition

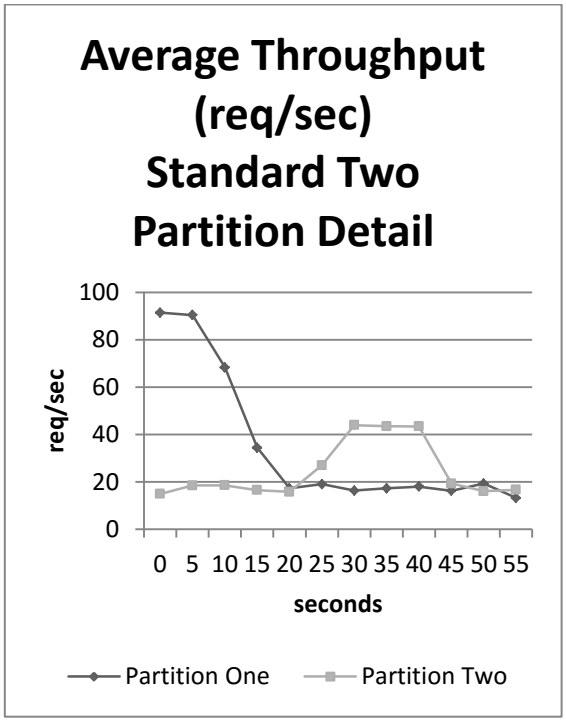

Figure 3.10: Throughput Two Partition

load. We also believe that this partitioning method is not as restrictive of the search indexes as traditional partitioning method. Each micropartition can have its own local index to speed up searches.

However, with respect to our clustering method, if too few clusters are selected for partitioning we may see imbalanced performance where one partition gives superior performance to the transactions involved and the other partition gives poor performance. Therefore, selecting more clusters allows for the allocation of the generated micropartitions in such a way that there is more evenly-balanced performance among nodes.

In the clusters in our experiment, we do not see balanced performance against the partitions in the two cluster experiment. In the two-cluster partitioning (one partition services 7 of the 15 queries; the other partition services the remaining 8 queries) we see greatly imbalanced latency and throughput performance (see Figures 3.11 and 3.12), although the average gives us higher overall performance.

However, with the five cluster selection and allocation, again running the appropriate queries against the appropriate clusters, we arrive at balanced performance on the nodes. We allocated three clusters on one node (25, 24, 21) and the remaining two clusters on another (22, 20). The details have been plotted in the graphs in Figures 3.13 and 3.14.

Graphs 3.13 and 3.14 show the latency and throughput values of the two nodes containing the five allocated micropartitions.

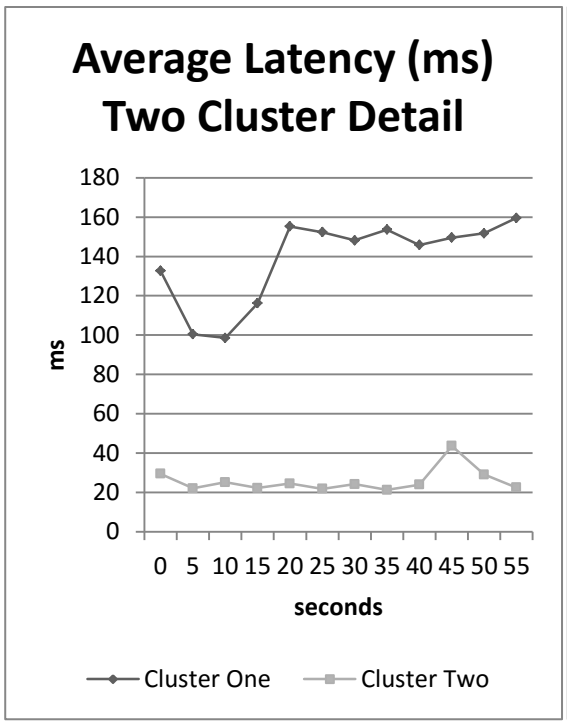

Figure 3.11: Latency Two Cluster

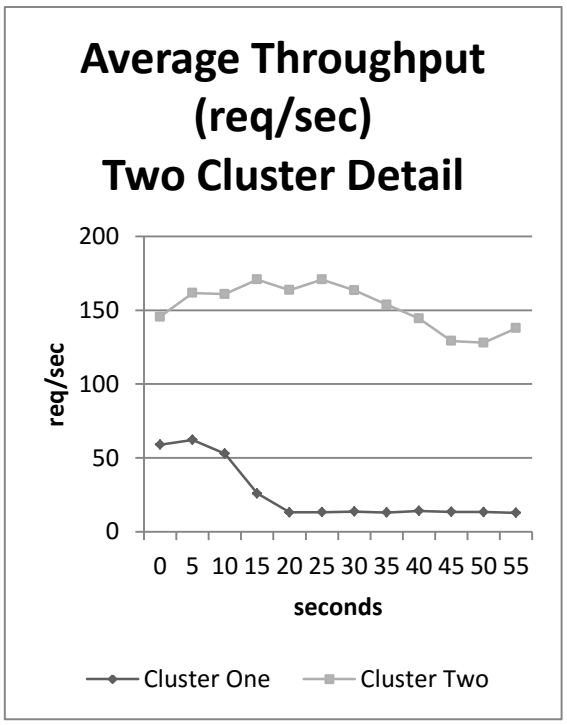

Figure 3.12: Throughput Two Cluster

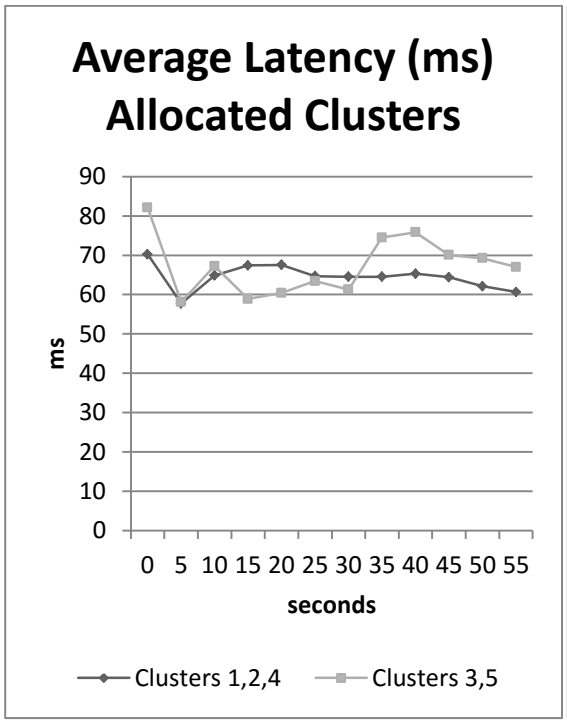

Figure 3.13: Latency Five Cluster

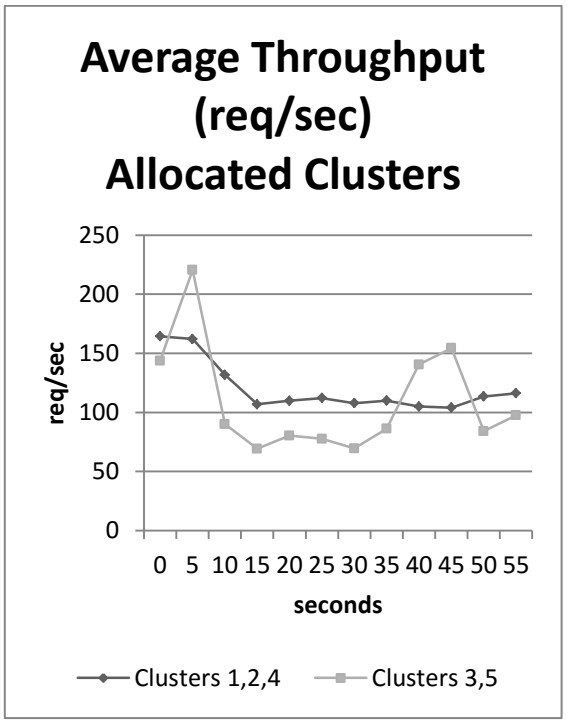

Figure 3.14: Throughput Five Cluster

A different approach for calculating query similarity, s, is to use query containment, in conjunction with query similarity, to assess the proximity of Q_i and Q_j. By assessing containment for query attributes, as well as assessing similarity, we can have another definition, $v^{'}$, rather than v where v' indicates complete similarity between the tuple sets of two queries if the tuples returned by one query in the pair are a subset of the tuples returned by the other query in the pair. Given two queries, Q_i and Q_j where the attributes A_i are a subset of the attributes A_j, we give a $v^{'}$ value of 1.0 because one set of attributes is fully contained within the other set of attributes as we see in the example in Figure 3.15.

$$A_i = a_1, a_3$$
$$A_j = a_1, a_2, a_3, a_4$$

Figure 3.15: Attribute Example

Containment and Similarity of Attributes Expression

The measurement, $v^{'}$, uses the Meet-Min coefficient to measure containment of and the similarity between the attributes in Q_i and Q_j. If the set of attributes of one query is a proper subset of the set of attributes of another query we consider this full containment and assign an attribute similarity value of 1.0. Otherwise the Meet-Min metric measures the similarity of two sample sets where there is no full containment. We merge the two abstract queries having the most similarity (value is closest to 1.0) with respect to their attributes.

For abstract queries Q_i and Q_j, $v^{'}$ evaluates containment and measures the similarity between the respective attribute sets, A_i and A_j.

$$v^{'} = |A_i \cap A_j| / min(|A_i|, |A_j|)$$

Using the same example as in Section 3.4, we see that there is not containment for the attributes in the projection statement. We apply the Meet-Min formula.

For example, given the following queries Q_1 and Q_2

Q_1: π c_discount, c_last, c_credit, w_tax, w_id,c_w_id σ w_id $< 5 \wedge$ c_w_id = w_id $\wedge$ c_w_id < 5

Q_2: π c_data1, c_w_id, c_d_id σ c_w_id ≤ 5

and their respective attribute sets

A_1: c_discount, c_last, c_credit, w_tax, w_id, c_w_id

A_2: c_data1, c_w_id

$v' = Meet - Min(Q_i, Q_j) = \frac{1}{min(2,6)} = \frac{1}{2} = 0.50$

Containment and Similarity of Query Condition Statements and Database Tuples

We replace h with measurement h'. By examining the predicates in Q_i and Q_j we may be able to determine, whether, for all instances, D, the resultant tuples for Q_i are fully-contained within the answer for Q_j.

The intuition is that query data for Q_i and Q_j should be found in the same partition if the answers for Q_i are fully contained within the answers Q_j, or visa versa, for any D. We can then say that one copy of the data satisfies both queries.

For those queries where the answer to one query is fully contained within the answer of another query, we will calculate an h' value of 1. For our example, we show C_i and C_j from Q_i and Q_j respectively, with each predicate expressed as p_i. In the example in Figure 3.16, we see clearly that the answer for Q_j is contained within the

answer for Q_i.

$$C_i = p_1$$
$$C_j = p_1 \wedge p_2 \wedge p_3$$

Figure 3.16: Condition Example One

Similarly, if $C_i \wedge C_j$ is not satisfiable, there are no tuples in any instance that can satisfy both C_i and C_j. The $h^{'}$ values for these query pairs will be 0, because the needed tuples can be in different partitions. There will be no need for concurrency control because the queries always access different tuples. In the example in Figure 3.17, we see that the answers for Q_i and Q_j are disjoint. Therefore, we calculate an $h^{'}$ value of 0.

$$C_i = p_1$$
$$C_j = !p_1 \wedge p_2 \wedge p_3$$

Figure 3.17: Condition Example Two

For other conjunctive queries, we cannot definitively say, based upon examining the syntax of the condition statements, that for every instance D, the answers of Q_i and Q_j will intersect. In other words, we cannot, by only looking at predicates of these queries, determine whether two queries will access the same tuples for any given database instance, as in the example in Figure 3.18.

$$C_i = p_1 \wedge p_2 \wedge p_3$$
$$C_j = p_1 \wedge p_4$$

Figure 3.18: Condition Example Three

For these queries, we need to examine the database instance, D.

The calculation, $h^{'}$, measures containment and similarity between the semantics of the condition statements, C_i and C_j or the tuples in the answer sets for Q_i and Q_j.

Summary of $h^{'}$ calculation:

Where C_i and C_j indicate answer sets that are always disjoint, $C_i \cap C_j = \emptyset$, $h^{'} = 0$.

Where C_i and C_j show containment, $C_i \subseteq C_j$ or $C_j \subseteq C_i$, $h^{'} = 1$.

Otherwise, given a database instance D, $h^{'} = |Q_i \cap Q_j|/min(|Q_i|, |Q_j|)$

For example, given the following abstract queries

Q_1: π c_discount, c_last, c_credit, w_tax, w_id, c_w_id σ w_id $< 5 \wedge$ c_w_id = w_id $\wedge$ c_w_id < 5

Q_2: π c_data1, c_w_id, c_d_id σ c_w_id ≤ 5

Because $Q_1 \subseteq Q_2$, $h^{'} = 1$

For the condition statements we see that there is full containment of Q_1 by Q_2. Notice that $h^{'}$ value is 1, indicating total containment. Therefore, in our example, the final proximity value is $p = 0.5 * 0.5 + 0.5 * 1.00 = 0.75$

Query containment is a well-studied area of research, originally studied for the purpose of query optimization. We use an algorithm that assesses disjunction, containment, or tuple overlap, to calculate $h^{'}$. If each predicate in Q_2 can replace a predicate in Q_1 and the execution of the modified Q_1 provides all the tuples for the old Q_1 then we have a containment mapping from Q_2 to Q_1. Refer to Figure 3.3 in order to understand the structure of each row in C_0.

Algorithm 8 $h^{'}$ Algorithm

Input: M_0, $C_0[i]$, $C_0[j]$, Q_i, Q_j
Output: $h^{'}$

$ct_1 = 0$,$ct_2 = 0$,$ct_3 = 0$,$ct_4 = 0$
for all col in C_0 **do**
  if $C_0[i][col] \cap C_[0][j][col == \emptyset$ **then**
    $h^{'} \Leftarrow 0$ **return**$h^{'}$
  else if $C_0[j][col] \implies C_0[i][col]$ **then**
    $ct_1 + +$
  else if $C_0[i][col] \implies C_0[j][col]$ **then**
    $ct_2 + +$
  else if $C_0[i][col] \neq \emptyset$ & $C_0[j][col] == \emptyset$ **then**
    $ct_3 + +$
  else if $C_0[i][col] == \emptyset$ & $C_0[j][col] \neq \emptyset$ **then**
    $ct_4 + +$
  end if
end for
if $(ct1 \neq 0\ And\ ct2 == 0\ And\ ct3 == 0)\ Or\ (ct1 == 0\ And\ ct2 \neq 0\ And\ ct4 == 0)$ **then**
  $h^{'} = 1$
else
  $h^{'} = |Q_i \cap Q_j|/min(|Q_i|, |Q_j|)$
end if**return**$h^{'}$

Chapter 4

Phase Two - Parallel Scheduling Framework For Workload

In this chapter, we assume that the web services are executing on parallel processors. We aim for a balanced workload across processors and to guarantee data consistency. In order to accomplish these aims we develop a process that orders the execution sequence of services, and calculates minimal delays, if necessary, to ensure correctness.

4.1 Introduction

Phase Two answers the following research question:

> How can we build a schedule for a parallel processing database that maximizes parallel execution of services while maintaining conflict serializability?

Our research is concerned with improving database query response time for system workloads. The overhead costs for workload query processing in parallel database environments can be very high. Chapter 2 describes, in detail, the costs involved in processing queries in a parallel database environment.

Our claim is that we can achieve a workload schedule that provides improved performance over traditional scheduling methods, for shared-memory and in-memory database systems. We improve efficiency by creating a schedule for a multiprocessor database that eschews the use of traditional correctness protocols while supporting parallel processing. Section 4.2 gives definitions required for this work. Section

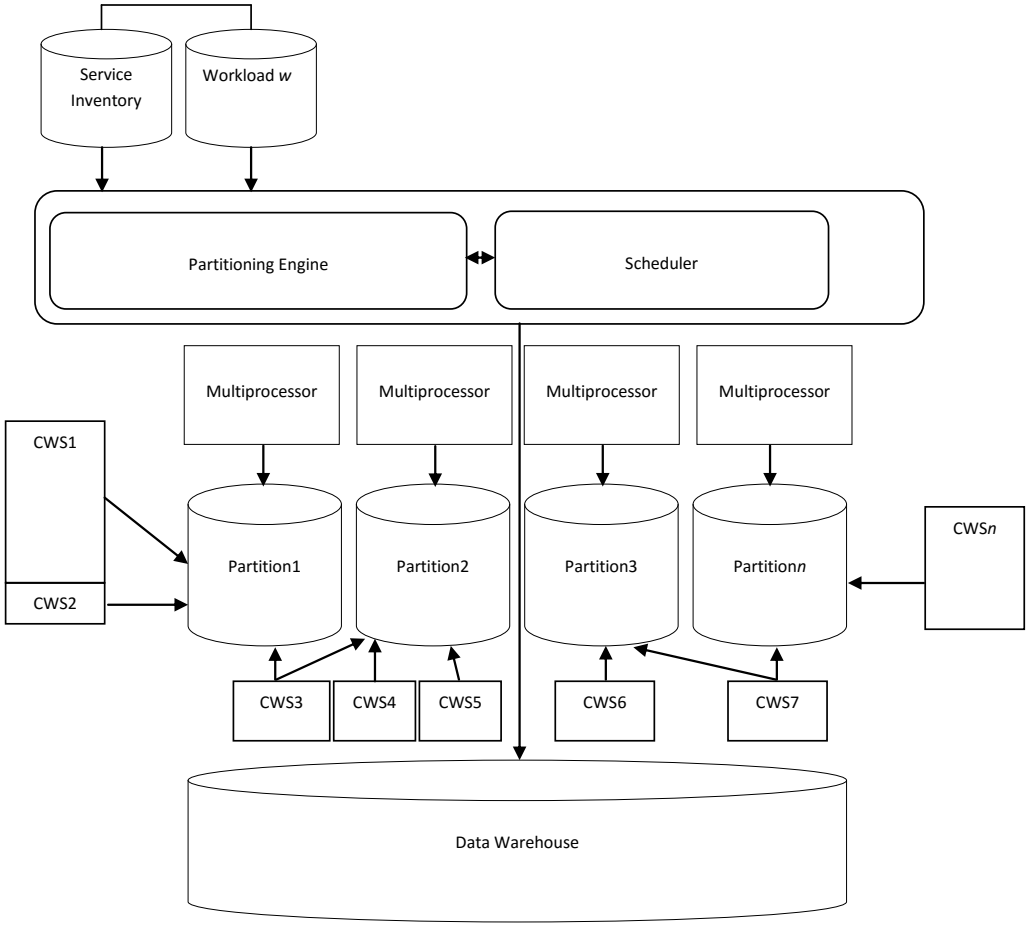

Figure 4.1: Framework for Phase II - Partitioning System

4.3, shows the scheduling procedure for a multiprocessor, single DBMS node. The scheduling procedure allocates web services from the complex web services, preserving correctness while maximizing parallel execution of the services. We describe how an algorithm that creates a schedule that preserves correctness and provides good parallelism.

4.2 Definitions

In Phase One, we defined an atomic web service, or WS, to be a transactional unit of processing, in a service-oriented environment, that executes one query. As a reflection of a realistic workload, not all transactions will be atomic because many transactions execute multiple queries. Therefore, we define a complex web service, or CWS, to be a transactional unit of processing that executes one or more queries. Queries within CWS may have precedence relationships. Queries between CWS may also execute in parallel. In addition to defining CWS, we also define a workload w, comprised of CWS.

Definition 5: Complex Web Service (CWS)
A complex web service is a partial order of atomic web services. The CWS composition is represented as a pair, $(\{WS_1, ..., WS_n\}, \leq)$ where, WS_i $(i = 1, ..., n)$ is an atomic web service and $\leq$ is the partial order relation.

Figure 4.2 shows a CWS as a DAG where each participating vertex in the graph represents a web service and a dashed, directed edge between two vertices represent the precedence relationship between a pair of web services. Vertexes unconnected by a path, or set of edges, may execute in parallel.

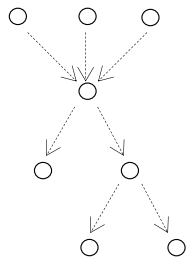

Figure 4.2: CWS_i Represented as a DAG

For our research, we define a system workload, w as $\{CWS_1, ..., CWS_r\}$. The workload w can also be viewed as a set of atomic web services, or WS and the partial order information for those WS.

Definition 6: Workload w

A workload is a set of complex web services. The workload, w, is represented as $\{CWS_1, ..., CWS_r\}$ where CWS_i $(i = 1, ..., r)$ is a complex web service.

The workload w can also be represented as

$(\{WS_1, ..., WS_l\}, \leq_1), ..., (\{WS_m, ..., WS_n\}, \leq_r)$

We calculate the total execution time of CWS. First, we calculate the processing cost, pc, for each WS in workload w from the CWS. In Phase I, we defined WS to be a relational algebra query, Q, where A is the set of attributes used in Q and $|Answer(Q)|$ is the number of tuples returned.

pc - Processing cost of a web service

For web service WS, pc calculates the

number of data items accessed.

$pc = A * |Answer(Q)|$

Definition 7: Graph, G_{WS}, for workload w $\{CWS_1, ..., CWS_r\}$ is defined as a partially-connected graph, $G_{WS} = (WS, A_p, A_c, n)$ where

- $WS = \bigcup_{i=1}^{r} WS \in CWS_i$ represents the graph vertices
- n labels each vertex with the pair $n_i = (WS_i, pc_i)$ where WS_i is the name of the web service and pc_i represents the processing cost of WS_i
- $A_p = \{a_{p1}, ..., a_{pq}\}$ is the set of directed edges, resulting from $\leq_i$ where $(i = 1, ..., r)$

- $A_c = \{a_{c1}, ..., a_{cr}\}$ is the set of undirected edges representing conflicting web services

In Figure 4.3, we represent a workload w as a graph, G_{WS}. In G_{WS}, a precedence edge is represented by a dashed arc. The undirected, conflict edges are represented as solid lines.

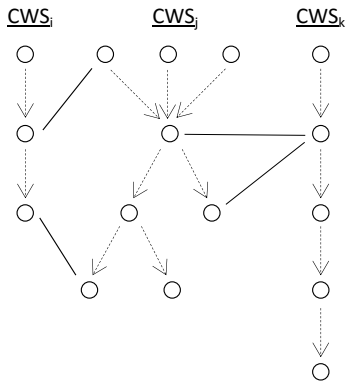

Figure 4.3: G_{WS} Representation of Workload w

The ordered conflict graph, $CWS_i \rightarrow_c CWS_j$, in Definition 8 and in Figure 4.4, represents the ordered conflict pair (CWS_i, CWS_j).

Definition 8: Ordered Conflict Graph, $CWS_i \rightarrow_c CWS_j$, $CWS_i, CWS_j \in w$
Let complex web services, CWS_i and CWS_j be components of a G_{WS}. We say that the "Ordered-Conflict Graph", denoted as $G_{i \rightarrow j}$, is a subgraph of G_{WS} satisfying the following conditions

- The nodes of $G_{i \rightarrow j}$ are over the nodes of CWS_i and CWS_j,
- The node labels in $G_{i \rightarrow j}$ are the same as in G_{WS},
- The edges, $A_{pii'jj'} = \{a_{p1}, ..., a_{pm}\}$, representing the precedence relation within CWS_i and within CWS_j in G_{WS}, are preserved, and,
- The conflicting, undirected edges, $A_{cij} = \{a_{c1}, ..., a_{cn}\}$, between conflicting operations $op_i \in CWS_i$ and $op_j \in CWS_j$ are replaced by a directed edge from op_i to op_j.

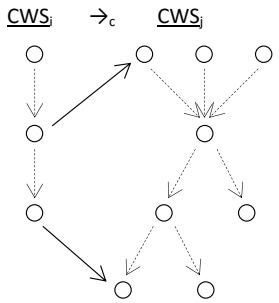

Figure 4.4: Example of Ordered Conflict Graph, $CWS_i \rightarrow_c CWS_j$

The topological conflict graph, $TO_i \rightarrow_c TO_j$, in Figure 4.5 represents the ordered conflict pair (CWS_i, CWS_j) where CWS_i is represented in topological order TO_i and CWS_j is represented in topological order TO_j, and where all conflicting operations of TO_i precede all conflicting operations of TO_j. The value, $prec$, is the sum of the processing costs, pc, of all web services that precede a WS in a TO.

$prec$ - total processing cost of web services preceding a WS in TO

Let there be the set of web services that precedes WS in a TO.

$WS_1, ..., WS_m$

Then, the $prec$ value of WS is

$$prec = \sum_{j=1}^{m} pc_j$$

Topological Conflict Graph, $TO_i \rightarrow_c TO_j$ of $CWS_i \rightarrow_c CWS_j$

is computed as

- Each vertex label, n_i in TO_i and n_j is extended from $n_i = (WS_i, pc_i)$, to be a triple, $n_i = (WS_i, pc_i, prec_i)$ where $prec_i$ is the sum of all $prec_1, ..., prec_{i-1}$
- Edges, $A_{pii'jj'}$ are preserved
- Edges, A_{cij} are preserved

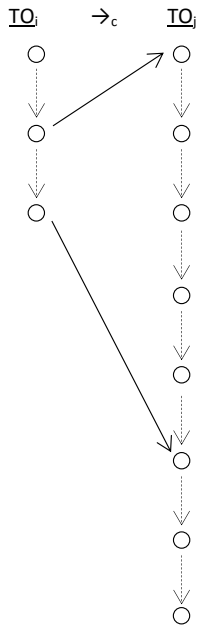

Figure 4.5: Example of Topological Conflict Graph, $TO_i \rightarrow_c TO_j$

Note, TO_i and TO_j both can be represented as an ordered list and the directed edges of A_{cij} can be represented as pairs of (n_i, n_j).

When dealing with a partial order CWS_j, we create a topological ordering, TO_j, as in Figure 4.6 (a) to (b), from which to calculate delay. However, different topological orderings, TO_j, can produce different delay calculation results, as can be seen in 4.6(b) and (c). The ordering in 4.6(b) is non-serializable. The precedence graph of $TO_i \rightarrow_c TO_j$ in Figure 4.6(b) is acyclic. The ordering in 4.6(c) is serializable. We want to find the topological orderings that produce a serializable schedule between an ordered conflicting complex web service pair, i.e., $TO_i \rightarrow_c TO_j$.

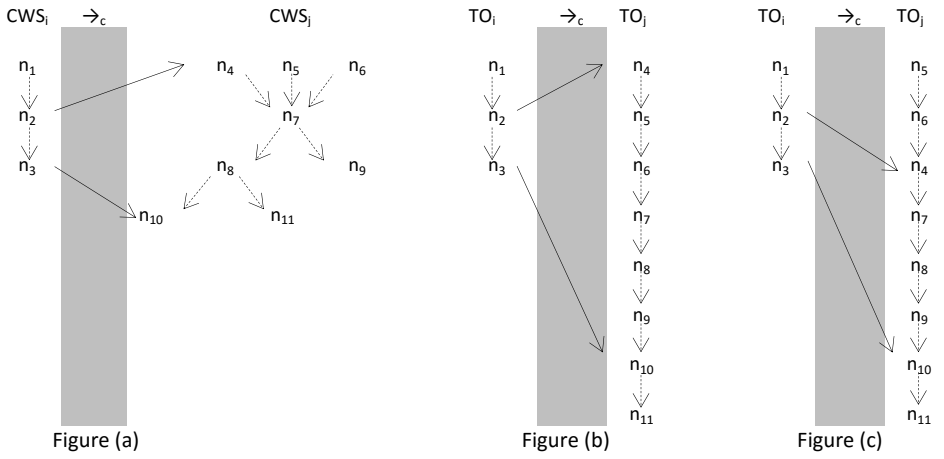

Figure 4.6: Processing CWS with Partial Order Information

Our goal is to develop a schedule for w, executing on a database with two processors, that maximizes parallel execution of services while preserving serializability. Given a workload w comprised of complex web services, we can expect to have conflicting operations between CWS. When executing multiple complex web services, we will show that the order in which the conflicting services are scheduled impacts both the correctness and efficiency of the execution of w.

Intuitively, to build the schedule we first develop an ordering analysis of conflicting web service pairs. This analysis determines the best topological orders for transactions and the best transaction order to preserve serializability for all transactions in w. Where serializability cannot be imposed by controlling the topologies of partial order CWS and by ordering the transactions in the scheduling queue, we impose a delay in order to guarantee serializability. We guarantee the least delay imposed by the transactions while satisfying serializability.

We represent an ordering of a conflicting complex web service pair as $TO_i \rightarrow_c TO_j$ where conflicting operations in transaction TO_i execute before their corresponding conflicting operations in TO_j. The symbol $\rightarrow_c$ indicates the order and the subscript c indicates that the operations of the TOs are in conflict.

Upon determining the best order of execution for conflicting complex web service pairs (either $TO_i \rightarrow_c TO_j$ or $TO_j \rightarrow_c TO_i$) we can use this information to build a schedule for w (See Figure 4.7). Using the ordering information, we build a schedule that supports parallelism while maintaining conflict serializability.

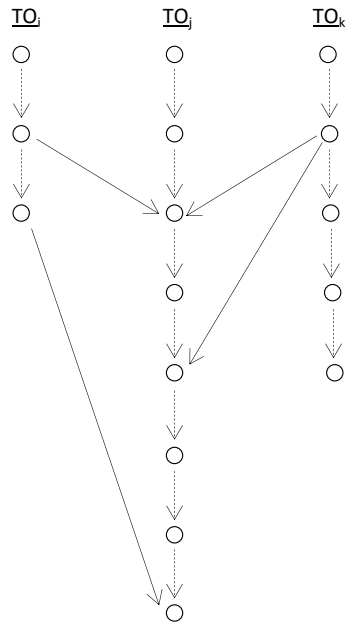

Figure 4.7: Ordering Information to Build Schedules for w

4.3 Scheduling

We select a preferred topological ordering of partial order complex web services to enforce serializability. We also select a preferred order of conflicting operations in transactions to control consistent execution of conflicting operations. If topological ordering and transaction ordering is not sufficient to enforce serializability, we estimate delay requirements for complex web services executing in parallel. Adding delays to a CWS execution will ensure that the parallel execution history is serializable. Note, that the topologies for each partial order complex web service, if not eliminating execution delays, will minimize execution delays.

We assume that the web services are executing on parallel processors. We aim for a balanced workload across processors and guarantee data consistency. For the first, ordering step, given ordered conflict graph, $CWS_i \rightarrow_c CWS_j$, we find the topological conflict graph, $TO_i \rightarrow_c TO_j$, such that there is minimum delay in the execution of TO_j. In Figure 4.8, we show TO_i and TO_j which are executing in parallel.

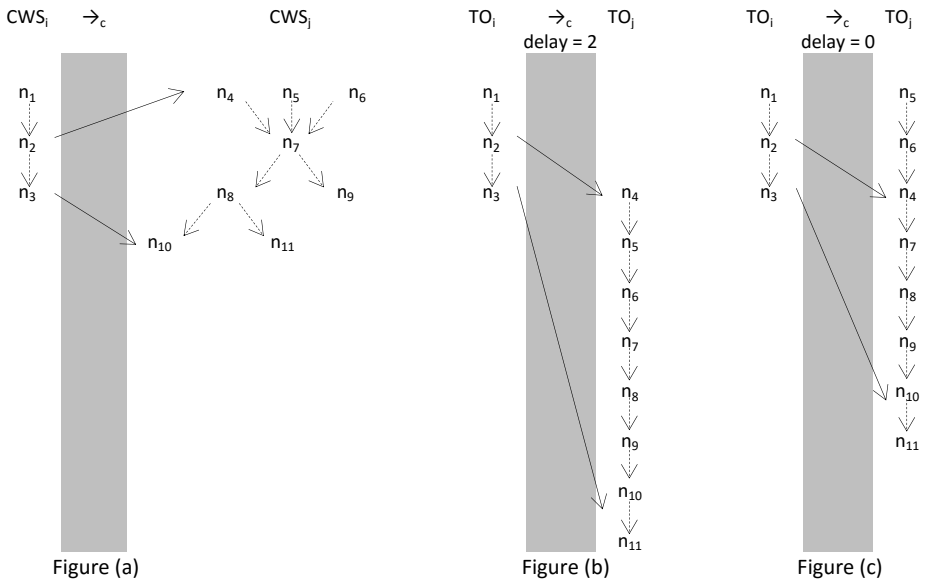

Figure 4.8: Delay d for Consistent Order of Operations for Conflicting CWS

Figure 4.8(b) shows parallel execution of TO_i and TO_j, for CWS_i and CWS_j depicted in Figure 4.8(a). Both complex web services start at the same time. Notice that serializability in (b) is preserved by introducing a delay. Notice that serializability is preserved in (c) by using a different topological ordering for CWS_j. Of course, we want a schedule with no delays but, if delays must be introduced, we want them to be minimal.

It is important to assess the delay of both $TO_i \rightarrow_c TO_j$ and $TO_j \rightarrow_c TO_i$ if there are conflicting operations between CWS_i and CWS_j. If there are no conflicts, then complex web services can be assigned to execute in parallel without problems. A conflicting complex web services pair, $\{CWS_i, CWS_j\}$ is a pair of complex web

services, CWS_i and CWS_j, where there is at least one conflicting operation between CWS_i and CWS_j.

To accomplish the ordering step, we calculate the delay that would be imposed on TO_j if $TO_i \rightarrow_c TO_j$.

Algorithm 9 CalculateDelay Algorithm

Input: $TO_i \rightarrow_c TO_j$
Output: d
1: $d \leftarrow 0$
2: **for all** pairs of conflict operations, $\{n_i : WS, n_j : WS\}$ where $n_i : WS \in TO_i$ and $n_j : WS \in TO_j$ **do**
3: $\quad d \leftarrow max(d, (n_i : prec + n_i : pc) - n_j : prec)$
4: **end for**
5: **return** d

Claim 1: Delay, d, calculated by Algorithm 9, when introduced prior to the execution of TO_j, is sufficient to guarantee that all conflicting operations of TO_i complete before the corresponding conflicting operations of TO_j.

Proof Sketch: By contradiction, assume that delay d, when introduced before the beginning of execution of TO_j, is not a sufficient delay to guarantee that a conflicting operation op_i of TO_i executes before the conflicting operation op_j of TO_j.

Let a single conflict web service pair, $\{n_i : WS, n_j : WS\}$, where $n_i : WS$ is in TO_i and $n_j : WS$ is in TO_j and $n_j : WS$ executes before $n_i : WS$ completes. But then, $n_j : prec + d$ must be less than $n_i : prec + n_i : pc$. That is, $n_j : prec + d < n_i : prec + n_i : pc$. Therefore, $d < (n_i : prec + n_i : pc) - n_j : prec$ But then, in line 3 of Algorithm 9, d's value would have been replaced by $(n_i : prec + n_i : pc) - n_j : prec$. This is a contradiction. □

An exhaustive search algorithm finds all topological orders for CWS_i and for CWS_j and calculates delay for each $TO_i^t \rightarrow_c TO_j^t$, where TO_i^t and TO_j^t are the current topologies to be tested. Algorithm 10 enumerates all candidates and calculates d for $TO_i^t \rightarrow_c TO_j^t$, returning optimal topologies for CWS_i and CWS_j in the form of $TO_i \rightarrow_c TO_j$.

Algorithm 10 TO_Exhaustive Algorithm

Input: $CWS_i \rightarrow_c CWS_j$
Output: $TO_i \rightarrow_c TO_j$

```
1: Initialize d to ∞
2: for all TO_i^t do
3:    for all TO_j^t do
4:       d^t ← calculateDelay(TO_i^t →_c TO_j^t)
5:       if d^t ≤ 0 then return TO_i^t →_c TO_j^t
6:       else if d^t < d then
7:          d ← d^t
8:          TO_j ← TO_j^t
9:          TO_i ← TO_i^t
10:      end if
11:   end for
12: end for
13: return TO_i →_c TO_j
```

Claim 2: Algorithm 10 finds the schedule, $TO_i \rightarrow_c TO_j$ of $CWS_i \rightarrow_c CWS_j$ that imposes least delay, d.

Proof Sketch: The proof trivially follows. □

The complexity of Algorithm 10 is $O(lk)$ where l is the number of all TOs of CWS_i and k is the number of all TOs of CWS_j. Therefore, to reduce the cost of selecting a schedule that guarantees minimal delay, our method builds the optimal topological orders of CWS_i and CWS_j for $CWS_i \rightarrow_c CWS_j$ by examining all valid combinations of sets of WS nodes rather than all valid combinations of individual WS nodes.

The basic intuition for building the topological order TO_j from CWS_j is to delay conflicting operations in order to reduce the likelihood that o_j is executed before the corresponding o_i in CWS_i. We do this by maximizing the number of web services, and thus the processing cost, preceding each conflict web service in CWS_j. When building the topological ordering TO_i from CWS_i, the intuition is to execute each conflicting web service as soon as possible. We achieve this by minimizing the number of web services, and thus the processing cost, preceding each conflict web service in CWS_i. Thus, for each conflicting web service pair $(n_i : WS, n_j : WS)$ we want the smallest $(n_i{:}prec + n_i{:}pc)$ and the largest $n_j{:}prec$.

If two conflict web services in CWS_i do not have a precedence relationship they can appear in either order in TO_i. The same, of course, holds true for $CWSj$ and TO_j. For each conflicting web service, we create a partial topological order as follows.

For CWS_i, given a conflicting operation, co_i, the partial topological order pTO_i is generated as:

Let operations $o_1, ..., o_l \in CWS_i$ such that o_k, $(k = 1, ..., l)$ must precede co_i and o_k is not a conflicting operation. pTO is then the topological order of $o_1, ..., o_k$ and is extended, at the end, with co_i.

For CWS_j, given a conflicting operation, co_j, the partial topological order pTO_j is generated as:

Let operations $o_1, ..., o_l \in CWS_j$ such that o_k, $(k = 1, ..., l)$ be the operations that are not preceded by co_j. The partial topological order pTO_j is a topological order over the operations, $o_1, ..., o_l$ and extended with co_j.

We use a combinatorial approach to select all possible combinations of partial topological orders so that we can find TO pairs with minimal delay. So, the heuristic is a brute-force algorithm operating at a higher level of granularity (i.e., partial TOs).

Flowchart 4.9, outlines logic for Algorithm 11. Algorithm 11 creates the $TO_i \rightarrow_c TO_j$ from $CWS_i \rightarrow_c CWS_j$ using the heuristic.

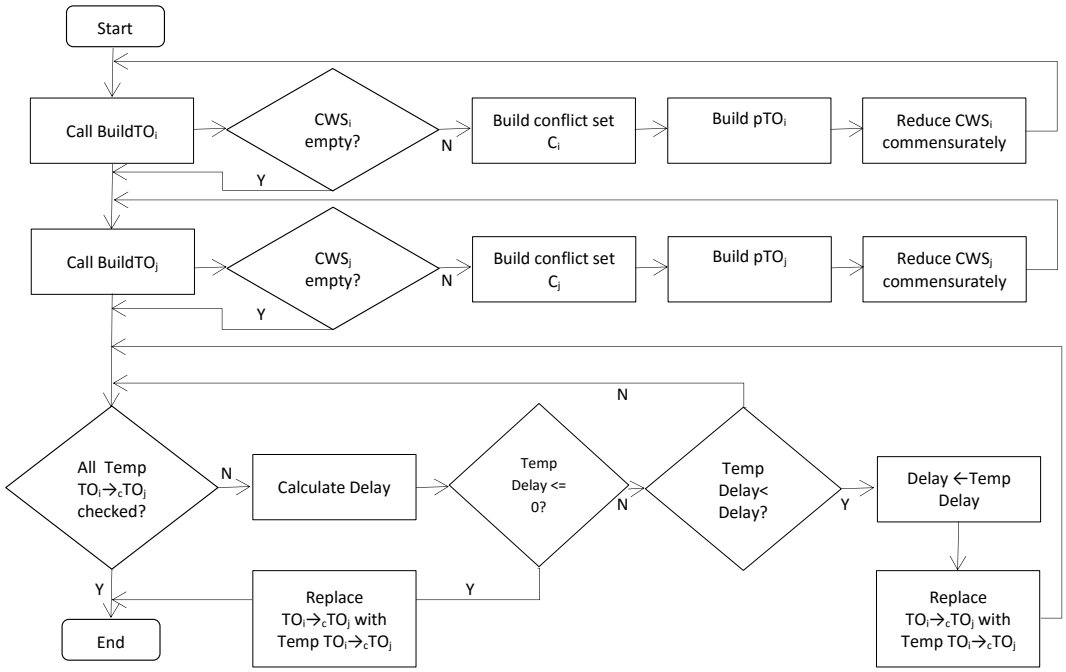

Figure 4.9: Flowchart for TO Algorithm

Example 4.10 shows the process of building a candidate TO_i from CWS_i and a candidate TO_j from CWS_j where $CWS_i \rightarrow_c CWS_j$.

The cost of executing Algorithm 11 is $O(c_k k_{level_1}...k_{level_m} c_l l_{level_1}...l_{level_n})$ where k is the number of conflict vertexes in CWS_i and l is the number of conflict vertexes in CWS_j. Each element in set $(k_{level_1}, ..., k_{level_m})$ represents the number of conflict vertexes in C_i at each iteration of $BuildTO_i$. The sub-equation, $k_{level_1} * ... * k_{level_m}$, represents the number of valid candidate TO_i built by combining blocks of web services comprised of conflict web services and their precedence web services.

Each element in $l_{level_1}, ..., l_{level_n}$ represents the number of conflict vertexes in C_j at each iteration of $BuildTO_j$. The sub-equation, $l_{level_1} * ... * l_{level_n}$, represents the number of valid candidate TO_j built by combining blocks of web services comprised

Algorithm 11 TO Algorithm

Input: $CWS_i \rightarrow_c CWS_j$
Output: Optimal $TO_i \rightarrow_c TO_j$

Initialize global lists L_i, L_j to $\emptyset$
Initialize d to ∞
Initialize $TO_i \rightarrow_c TO_j$, $TO_i^t \rightarrow_c TO_j^t$
$L_i \leftarrow \text{BuildTO}_i(CWS_i, TO_i)$
$L_j \leftarrow \text{Build}TO_j(CWS_j, TO_j)$
for all TO_i^t *in* L_i **do**
  for all TO_j^t *in* L_j **do**
    Add conflict edges, A_{cij}, to $TO_i^t \rightarrow_c TO_j^t$
    $d_t \leftarrow \text{calculateDelay}(TO_i^t \rightarrow_c TO_j^t)$
    if $d_t \leq 0$ **then return** $TO_i^t \rightarrow_c TO_j^t$
    else if $d_t < d$ **then**
      $d \leftarrow d_t$
      $TO_j \leftarrow TO_j^t$
      $TO_i \leftarrow TO_i^t$
    end if
  end for
end for
return $TO_i \rightarrow_c TO_j$

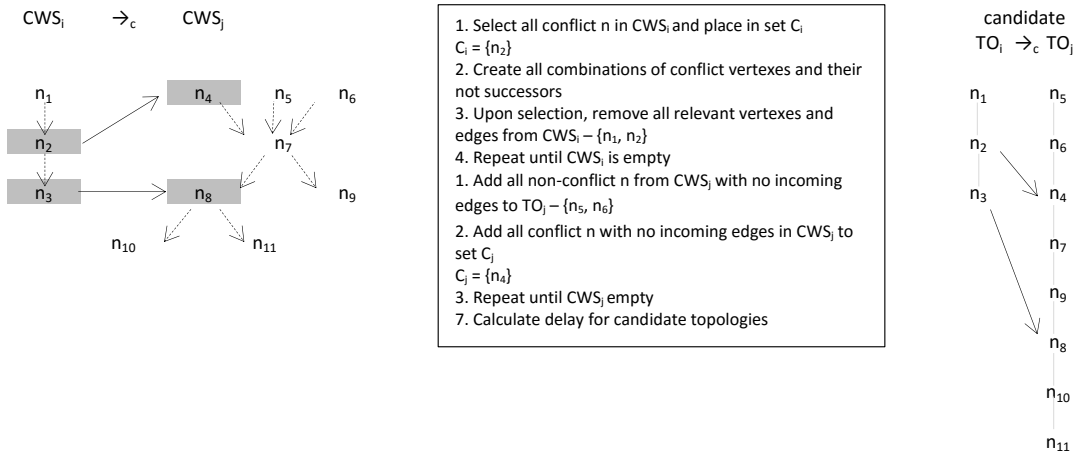

Figure 4.10: Building TO_i from CWS_i and TO_j from CWS_j for $CWS_i \rightarrow_c CWS_j$

Algorithm 12 BuildTO$_i$ Algorithm

Input: CWS_i, TO_i
Output: TO_i

```
1: Initialize C_i ← {∅}
2: if CWS_i not empty then
3:    CWS_i' ← CWS_i
4:    TO_i' ← TO_i
5:    for all n_i:WS where n_i:WS ∈ CWS_i and n_i:WS has no incoming edge do
6:       if n_i:WS is a conflict vertex then
7:          C_i ← C_i ⋃ n_i:WS
8:       else
9:          Insert n_i at end of TO_i
10:         Remove n_i and its edges from CWS_i
11:      end if
12:   end for
13:   for all n_i:WS ∈ C_i do
14:      CWS_i ← CWS_i'
15:      TO_i ← TO_i'
16:      CWS_i_subtree ← build subtree from CWS_i of all precedents to n_i:WS plus
    n_i:WS
17:      for all n_i:WS ∈ CWS_i_subtree do
18:         Insert n_i:WS at end of TO_i
19:         Remove n_i:WS and its edges, A_pii', from CWS_i
20:      end for
21:      BuildTO_i(CWS_i, TO_i)
22:   end for
23: end if
24: if C_i is empty then
25:   Insert TO_i to end of L_i
26: end if
```

of conflict web services and their precedence web services. The constants c_k and c_l represent the cost of searching CWS_i and CWS_j, at each iteration, for the subsets of non-conflicting web services to be paired with each conflict web service.

Claim 3: Algorithm 11 returns valid topological orderings, TO_i and TO_j for CWS_i and CWS_j in $CWS_i \rightarrow_c CWS_j$.

Algorithm 13 BuildTO$_j$ Algorithm

Input: CWS_j, TO_j
Output: TO_j

```
1: Initialize C_j ← {∅}
2: if CWS_j not empty then
3:    for all n_j:WS where n_j:WS ∈ CWS_j and n_j:WS has no incoming edge do
4:       if n_j:WS is a conflict vertex then
5:          C_j ← C_j ⋃ n_j:WS
6:       else
7:          Insert n_j:WS at end of TO_j
8:          Remove n_j:WS and its edges from CWS_j
9:       end if
10:   end for
11:   for all n_j:WS ∈ C_j do
12:      CWS_j' ← CWS_j
13:      TO_j' ← TO_j
14:      Insert n_j:WS at end of TO_j
15:      Remove n_j:WS and its edges from CWS_j
16:      BuildTO_j(CWS_j, TO_j)
17:      CWS_j ← CWS_j'
18:      TO_j ← TO_j'
19:   end for
20: end if
21: if C_j is empty then
22:   Insert TO_j at end of L_j
23: end if
```

Proof Sketch:

For this, we show that every order chain TO_i and TO_j is valid. That is, they cannot violate the precedence orders defined in CWS_i and CWS_j. In Algorithms 12 and 13 remove vertexes and edges from CWS_i and CWS_j respectively, of those vertexes that have no incoming edges. The respective topological orders are built by using the selected vertexes.

Each iteration of Algorithm 12 removes vertexes with no incoming edges and with a single common successor (removing the outgoing edges of those vertexes also) adding each vertex, as encountered, to the tail of TO_i. The single common successor is then added to TO_i and that vertex, and it's outgoing edges are removed from CWS_i.

Each iteration of Algorithm 13 removes vertexes with no incoming edges of a set of successors (removing outgoing edges of those vertexes also). A successor is added to TO_j and that vertex, and its outgoing edges are removed from CWS_j.

Each iteration of the Algorithms 12 and 13 preserve the topological order TO_i and TO_j respectively. □

Claim 4: Given $CWS_i \rightarrow_c CWS_j$, Algorithm 11, builds topologies, TO_i and TO_j that result the minimum delay, d, for $TO_i \rightarrow_c TO_j$.

Proof Sketch:

To assist in the analysis, we think of the layout of TO_i as a series of web service vertex blocks, which we call pTOs (for partial TO). Each pTO_i contains a conflict WS_i as its last vertex, preceded by all vertexes that precede WS_i in CWS_i. A pTO may contain only the conflict vertex. Figure 4.11 shows an example of this structure.

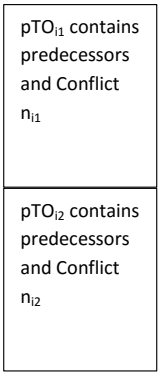

Figure 4.11: Example TO_i with pTOs

To assist in the analysis, we think of the layout of TO_j as a series web service vertex blocks, which we call pTOs. Each pTO_j contains one conflict WS_j as its last vertex, preceded by all vertexes that are not successors to a subset of conflict ver-

texes from CWS_j. A pTO may contain only the conflict vertex. Figure 4.12 shows an example of this structure.

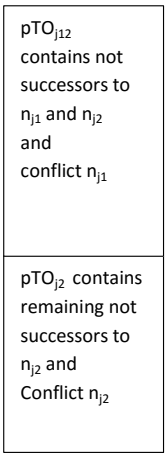

Figure 4.12: Example TO_j with pTOs

Every pTO, is treated as a single entity by Algorithm 11 performing a combinatorial search for optimal delay value at the pTO level. Optimal result is trivially true at this level. Next, we show that any move of a node within a pTO or to another pTO will result in either an invalid topology or a higher delay calculation.

In TO_i, from $TO_i \rightarrow_c TO_j$, changing TO_i to $TO_{i'}$ by moving one or more nodes in TO_i, results in the same or higher d for $TO_{i'} \rightarrow_c TO_j$ or results in an invalid topology $TO_{i'}$.

For TO_i, if a node within a block, pTO_{i1}, is moved elsewhere within pTO_{i1} this results in an invalid topology, $TO_{i'}$ because of change of precedence order.

If n_{i1} is moved into another block, pTO_{i2}, a block which comes after pTO_{i1} in TO_i, then $TO_{i'}$ is invalid because a precedence relationship has been broken.

If n_{i2} is moved out of pTO_{i2} into another pTO_{i1}, a block which appears earlier in TO_i, the resulting $TO_{i'}$ is an invalid topology, if conflict n_{i2} is the node that has been

moved. Otherwise, $TO_{i'}$ is a valid topology. In this case, d is increased for conflict relationship that involves n_{i1} with no change in d the conflict associates with n_{i2}.

In TO_j, from $TO_i \rightarrow_c TO_j$, changing TO_j to $TO_{j'}$ by moving one or more nodes in TO_j, results in the same or higher d for $TO_i \rightarrow_c TO_{j'}$ or results in an invalid topology $TO_{j'}$.

For TO_j, if a node within a block, pTO_{j1}, is moved elsewhere within pTO_{j1} this may or may not result in an invalid topology, $TO_{j'}$ because of change of precedence order. If $TO_{j'}$ remains a valid topology there is no change in d associated with n_{j1}.

If n_{j1} is moved into another block, pTO_{j2}, a block which comes after pTO_{j1} in TO_j, then $TO_{j'}$ then $TO_{j'}$ may or may not be a valid topology. If it is a valid topology, the d associated with n_{j1} is decreased.

If n_{j2} is moved out of pTO_{j2} into another pTO_{j1}, a block which appears earlier in TO_j, the resulting $TO_{j'}$ is an invalid topology because we are moving a successor node to some conflict node appearing after pTO_{j1}.

□

Chapter 5

Conclusions and Future Work

There are limiting factors to the scale out of transactional, distributed databases. One of the key limiting factors is the overhead associated with concurrency control in a distributed environment. The goal of concurrency control in a distributed database environment is to ensure global serializability. challenge to enforcing global serializability is the added cost of communication. The communication is necessary to establish the correct application of data updates for simultaneously executing transactions and the correct updating of replicas.

The main methods of concurrency control in a global environment have been the use of established mechanisms that were developed for a centralized environment. These mechanisms, such as strict, strong two-phase locking, have proven to be costly as distributed databases scale-out. Snapshot isolation, an implementation technique for multi-version concurrency control, and global commitment ordering, have proven to be useful in the reducing the costs of maintaining global consistency.

In Phase One is that we have addressed the overhead cost problem of scaling out transactional databases. In Phase One, we developed a method of data partitioning that is driven by query information from a system workload. In this phase of our development, we analyzed and addressed the partitioning of data for web services that have only one query each. We have shown that our method of partitioning improves performance over standard range partitioning techniques.

Because of the limiting factors to the scale out of transactional, distributed databases and because of development of commodity multi-processor database sys-

tems, there has been a re-examination of economical scale-up of database systems. Multiprocessor database systems can now process transactions at a faster rate. Therefore, all processing is done locally, thus localizing serialization of transactions. However, is has been found that the use of established serialization mechanisms on multiprocessor systems actually prevents efficient exploitation of the multi processor system [97]. Current concurrency control protocols were developed for uniprocessor systems. The art of scaling involves identifying the bottlenecks in a system and finding methods to avert those bottlenecks. Adding processing units to a database system is an example of hardware vertical scaling and averts the bottleneck of a single processor on a system. Creating in-memory databases, that also utilize either multiple processors or multiple cores, is a reworking of database systems to avert the I/O bottleneck to disk storage.

Locks are typically required for concurrency control and usually require hardware support for efficient implementation. Proper support for locks in a multiprocessor environment is very expensive incurring substantial synchronization issues. Optimistic methods of concurrency control have been used in multiprocessor systems to reduce the synchronization messages required to ensure data correctness.

In-memory database systems must also use multiprocessors or multicores. Because the single processor bottleneck and the I/O bottleneck to disk storage have been averted, there are of course, new bottlenecks to contend with. Traditional concurrency control mechanisms do not work well with multiprocessor, especially in-memory multiprocessor databases. Any kind of locking is the new bottleneck for systems with such high throughput.

In Phase Two we address the overhead cost problem of scaling up transactional database systems. Phase Two introduced calculated minimal delays for transactions in a multi-processor system. By ordering transactions and/or introducing minimal delays when sending conflicting transactions to multiple processors, we ensure correct

execution of conflicting transactions. Thus, concurrency control is implemented by ordering of transactions and the introduction of minimal delays, rather than with the use of locking mechanisms.

5.1 Future Work

An issue for future research is to extend the partitioning method to work, not only for atomic web services, WS, but for complex web services, CWS, as well. One method could be to partition data for CWS would be to group, by data similarity, at a higher granularity, at the CWS level. This entails grouping the CWS on their similarity level, which we call P, the sum of all p for individual conflicting operations between a given $\{CWS_i, CWS_j\}$.

A second area for research would be to examine the ways in which standard partitioning could be used in conjunction with the partitioning method developed in Phase One. One reason for the integration of standard partitioning methods is the inherent replication of the Phase One partitioning process. Another reason is Phase One partitioning could result in no horizontal partitioning due to the nature of the selection statements.

First, there can be inherent replication in the partitions created from our partitioning method. The partitioning technique from Phase One is not a true partitioning in that there can be a overlap in data requirements for the resultant partitions. Depending upon the nature of the workload, w, the resulting data overlap for the partitioning could be nothing, due to sufficient variability of data used by conflicting complex web services. The partitioning, at the other extreme, could result in full replication of data, because every transaction in the workload requires the same data.

A solution for this would be to integrate standard-partitioning methods with the partitioning method outline in Phase One. There may be a certain subset of transac-

tions that partition well using the method outlined in Phase One. The process that selectively partitions the data for a workload using a number of partitioning options would be a fruitful area of research.

Second, the Phase One method for identifying the tuples required by a WS is to examine the selection statement within the query. For our analysis, we removed all user-defined conditions, a user-defined condition being a condition where the information is supplied at run-time and assigned to a variable in the condition. Re-incorporating user-defined conditions implies that all tuples must be provided for that query because the tuple requirements are unknown until run-time.

In this case, it would be useful to develop a process to layering standard partitioning atop the Phase One partitioning method. Application of multiple partitioning methods groups the transactions using Phase One clustering technique without the transactions that contain user-defined conditions. The data that is required by the transactions with the user-defined conditions are retrofitted into the partitions.

Another reason for such an integration approach is to re-introduce intra-query parallelism into the system as the partitioning method in Phase I only addresses inter-query parallelism. Performing standard partitioning atop our method would achieve both goals.

The third area of research would develop a framework for dynamically updating the partitions and dynamically managing replicas. Dynamic updating of the partitions is a response to changes in the transactions of the workload. The process would require mechanisms for updating transaction groups as new complex web services are scheduled and old complex web services leave the system [32] [76] [113] [43] [42] [24].

Dynamic updating is also a response to changes in the data itself. It is important to develop a process that would efficiently update the database partitions [21] [101] [41] [104] [31] [59] [6] [94] [47]. A process for updating the partitioning as data is

updated, deleted, and added is a fruitful area of research.

In Phase One, the partitioning method places a single copy of data on a nodes, for write-oriented transactions that conflict. However, with remaining transaction types (where transactions are globally read-only) there can be many copies of a data item. If transactions enter the system, that use the data that was previously used only for read-only transactions, there may need to be a major re-partitioning of the data. This is an important scenario, and the re-partitioning process would need to be addressed.

As the volume of users accessing a transaction increasing or decreases, it is necessary to have an efficient and automatic replication process to spread user access across multiple nodes [112] [82]. A key advantage of the data partitioning framework developed in Phase I is that it is possible to create replicas only for those transactions that experience a surge in user activity. All other transactions are unaffected by the use of such targeted replication.

We have developed a method of data partitioning that is driven by query information from a system workload. In this phase of our development, we have analyzed and addressed the partitioning of data for web services that have only one query each. We have found that our method of partitioning shows improved performance over standard range partitioning techniques.

In future work we would apply a variation of this partitioning technique to workloads comprised of web services having multiple queries and we demonstrate how we can effectively localize processing for these types of services. We show how this reduces the cost of distributed web service processing.

In future work we would also demonstrate how we can efficiently reallocate data as the data in the partitions changes and as the web services within the workload also change.

In Phase Two we optimized the topologies of two complex web services in order

to ensure the minimum delay needed to preserve correctness when the transactions execute in parallel. There are many avenues of research that can be conducted to develop this method of concurrency control.

The first area of research would be to extend the concept of finding minimal delay, preferably no delay, to ensure correctness for complex web services executing on k processors. Another area of research would be to integrate topological ordering and the introduction of delays to complex web services that have interleaved execution on a single processor. A third area of research would be the preservation of correctness using topological ordering and delays where web services, within an individual complex web service are allocated to different processors in order to take advantage of the internal possibilities for parallelism within a single complex web service.

A fourth area of research would be to analyze changing delay requirements needed to preserve correctness in a dynamic environment, where new service requests continuously enter the system. A fruitful area of research would be to develop a procedure for dynamic topology switching in order to ensure that any simultaneously executing complex web services are using optimized topologies.

Integrating the partitioning framework with the scheduling framework is also a beneficial area of research and analysis. The HEFT, or heterogeneous earliest finish time algorithm, is a heuristic that schedules a set of dependent tasks onto a set of heterogeneous systems [29]. HEFT uses precedence relationships and communication costs to determine assignments of tasks. We could integrate the similarity measurement into an algorithm such as HEFT determine the assignment of tasks to nodes.

Bibliography

[1] D. Abadi. Problems with cap, and yahoo's little known nosql system. Technical report, Department of Computer Science, Yale University, April 2010.

[2] D. Abadi, S. Harizopoulos, S. Madden, and M. Stonebraker. Horizontica-a new approach to oltp data bases. Technical report, MIT, 2007.

[3] D. Abadi, S. Harizopoulos, S. Madden, and M. Stonebraker. Relational cloud: The case for a database service. Technical report, MIT, 2010.

[4] D. Abadi, S. Harizopoulos, S. Madden, and M. Stonebraker. Vertica-an hp compnay. Technical report, Hewlett-Packard, 2012.

[5] D. Abadi, S. Harizopoulos, S. Madden, and M. Stonebraker. Voltdb. Technical report, Brown University, 2012.

[6] S. Agarwal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. *SIGMOD*, pages 359–370, June 2004.

[7] F. Ahmad and A. Sarkar. Scheduling of composite services in multi-cloud environment. *International Conference of Grid and Cloud Computing and Applications*, 2015.

[8] S. Albers. Online algorithms: A survey. *Mathematical Programming*, pages 3–26, July 2003.

[9] M. Antonioletti, C. B. Aranda, O. Corcho, M. Esteban-GutiÃĭrrez, A. GÃşmez-PÃĭrez, I. Kojima, S. Lynden, and S. M. Pahlev. *WS-DAI RDF(S): Introduction, Motivational Use Cases and Terminology*. W3C, December 2009.

[10] M. Antonioletti, M. Atkinson, A. Krause, S. Laws, S. Malaika, N. W. Paton, D. Pearson, and G. Riccardi. *Web Services Data Access and Integration: the Core (WS-DAI) Specification, Version 1.0 Full Recommendation*. W3C, April 2012.

[11] M. Antonioletti, S. Hastings, A. Krause, S. Langella, S. Lynden, S. Malaika, and N. W. Paton. *Web Services Data Access and Integration: The XML Realization (WS-DAIX)Specification, Version 1.1.* W3C, May 2009.

[12] P. Apers. Data allocation in distributed database systems. In *ACM Transactions on Database Systems*, pages 263–304, September 1988.

[13] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore- providing scalable, highly available storage for interactive services. *Fifth Biennial Conference on Innovative Data Systems Research*, pages 223–234, 2011.

[14] R. Barack and F. Budinsky. *OASIS-Service Data Objects for Java, Version 3.0.* OASIS, November 2009.

[15] L. Bellatrech, K. Karlapalem, and M. Mohania. *Data Warehousing Web Engineering.* InfoSci-Booksl, 2002.

[16] P. Berkhin. Survey of clustering mining techniques. Technical report, Accrue Software Inc., September 2009.

[17] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[18] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[19] M. Brooker. Cap and pacelc: Thinking more clearly about consistency. Technical report, Amazon Elastic Block Store, July 2014.

[20] R. Cattell. Scalable sql and nosql data stores. In *SIGMOD Record*, December 2010.

[21] M. Charikar, C. Chekuri, T. Feder, and R. Matwani. Incremental clustering and dynamic information retrieval. *Proceedings of the 29th Annual Symposium on Theory of Computing*, pages 626–635, 1997.

[22] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technologies. *ACM Sigmod Record*, 26(1):65–74, March 1997.

[23] T.C. Cheng and S. Podolsky. *Just-in-Time Manufacturing: An Introduction, Second Edition*. Chapman-Hall, 1996.

[24] C. Chinrungrueng and C. Sequin. Optimal adaptive k-means algorithm with dynamic adjustment of learning rate. *IEEE Transactions for Neural Networks*, 6:157–169, January 1995.

[25] Clustrix. The scalable database: Why sharding doesn't work. Technical report, Clustrix, 2014.

[26] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!âĂŹs hosted data serving platform. *Proceedings of the VLDB Endowment*, pages 1277–1288, 2008.

[27] C. Curino, D. E. Difallah, A. Pavlo, and P. Cudre-Maroux. Oltp benchmark workloads. Technical report, Consortium of Brown University and MIT, 2012.

[28] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. In *Proceedings of the VLDB Endowment*, 2010.

[29] M. I. Daoud and N. Karma. A high performance algorithm for static task scheduling in heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 68:399–409, 2008.

[30] S. Das, D. Agrawal, and A. El Abadi. G-store: A scalable data store for transactional multi key access in the cloud. *Proceedings of the 1st ACM symposium on Cloud computing*, pages 163–174, 2010.

[31] S. Das, D. Agrawal, and A. E. Abbadi. Elastras: An elastic transactional datastore in the cloud. *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, 2009.

[32] S. Dasgupta and P. M. Long. Performance guarantees for hierarchical clustering. In *Journal of Computer System and Sciences*, volume 70, pages 555–569, June 2005.

[33] C. Dutra de Aguar Ciferri and F. da Fonseca de Souza. Distributing the data warehouse. In *Proc.XV.SBBD Symposium*, 2000.

[34] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. Technical report, Google, 2011.

[35] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: AmazonâĂŹs highly available key-value store. *ACM Computing*, pages 205–220, October 2007.

[36] E. Deelman, K. Vahi, M. Rynghe, and G. Juve. Pegasus in the cloud: Science automation through workflow technologies. *Internet Computing, IEEE*, 20, January 2016.

[37] D. J. Dewitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 1–8, 1984.

[38] D. A. DiMello and V. S. Ananthanarayana. A review of dynamic web service description and discovery techniques. *2010 First International Conference on Intelligent Computing*, pages 246–251, 2010.

[39] J. Eastman. Canopy clustering. Technical report, Apache, 2012.

[40] M. Edwards. Service data object overview. Technical report, Apache, 2011.

[41] A. J. Elmore. *Elasticity Primitives for Database as a Service*. PhD thesis, University of California Santa Barbara, Distributed Systems Lab, 3 2014.

[42] D. Eppstein. Fast hierarchical clustering and other applications of dynamic pairs. In *Journal of Experimental Algorithmics*, pages 1–10, January 2000.

[43] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, pages 669–696, September 1997.

[44] T. Erl. *Service-Oriented Architecture(SOA): Concepts, Technology, and Design*. Prentice-Hall, 2005.

[45] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, 2015.

[46] J. M. Faleiro, A. Thompson, and D. J. Abadi. Lazy evaluation of transactions in database systems. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 15–26, 2014.

[47] A. Fiat, Y. Mansour, A. Rosen, and O. Waarts. Competitive access time via dynamic storage rearrangement. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 392–403, October 1995.

[48] T. Forell, D. Milojicic, and V. Talwar. Cloud management: Challenges and opportunities. *2011 IEEE International Parallel and Distributed Processing Symposium*, pages 881–889, 2011.

[49] M. Gadgil. Simple and practical: Ideas and practical tips for software professionals. Technical report, Mahesh Gadgil Blog, January 2011.

[50] H. Garcia-Holina and D. Barbara. The cost of data replcation. *SIGCOMM 81 Proceedings of the seventh symposium on Data communications*, pages 193–198, 1981.

[51] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition tolerant web services. *ACM SigAct News*, 33:51–59, June 2002.

[52] K. Grolinger, W. A. Higashino, A. Tiwari, and M. A. Capretz. Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing*, 2(22), October 2013.

[53] Oracle Group. Oracle:partitioning in data warehouses. Technical report, Oracle, 2005.

[54] P. Helland. Life beyond distributed transactions: An apostate's view. Technical report, Amazon, January 2007.

[55] Y. Huang and J. Chen. Fragment allocation in distributed database design. *Journal of Information Science and Engineering*, 17:491–506, 2001.

[56] A. Jain. *Algorithms For Clustering Data*. Prentice-Hall, 1988.

[57] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. In *ACM Computing Surveys*, pages 264–323, March 1999.

[58] A. K. Jain, A. Topchy, M. H. Law, and J. Buhmann. Landscape for clustering algorithms. *Proceedings IAPR International Conference on Pattern Recognition*, pages 1–4, June 2004.

[59] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. *Springer Verlag*, pages 65–80, 2012.

[60] E. G. Coffman Jr., J. Csirik, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms: Survey and classification. *Handbook of Combinatorial Optimization*, pages 455–531, 2013.

[61] R. Kathari and D. Pitts. On finding the number of clusters. *Pattern Recognition Letters*, 20:405–416, 1999.

[62] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[63] J. LaCouture. Introducing database microsharding and oinky. Technical report, Oinky, March 2012.

[64] A. Lakshman and P. Malik. Cassandra - a decentralized, structured storage system. *ACM Computing*, 2009.

[65] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High performance concurrency control mechanisms for in-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, 2011.

[66] F. Lehman, C. Fehling, R. Meitzner, A. Nowak, and S. Dustdar. Moving applications to the cloud: An approach based on application model enrichment. *International Journal of Cooperative Information Systems*, 20(3):307–356, 2011.

[67] J. K. Liker. *The Toyota Way.* McGraw-Hill, 2004.

[68] X. Lin, M. Orlowska, and Y. Zhang. On data allocation with minimal overall communication costs in distributed database design. *IEEE Distributed Database Design*, pages 539–544, 1993.

[69] A. Liu, D. Batista, and M. Alomari. A survey of large scale data management approaches in cloud environments. *Communications Surveys and Tutorials*, pages 311–336, 2011.

[70] R. Malik, N. Bisht, and P. Mishra. Spw: Scheduling and positioning of webservices. *International Journal of Scientific and Engineering Research*, 4, November 2013.

[71] S. Marston, Z. Li, S. Bandyopadhyay, and A. Ghalsasi. Cloud computing: The business perspective. *Decision Support Systems*, 51(1):176–189, April 2011.

[72] M. Meyer. Riak handbook. Technical report, Basho, 2011.

[73] G. Milligan and M. Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50:159–179, 1985.

[74] H. L. Morgan and K. D. Levin. Optimal program and data locations in computer networks. *Management Science and Operations Research*, pages 315–322, 1977.

[75] F. Murtagh and P. Contreras. Methods of hierarchical clustering. *Cornell University Library*, pages 1–21, May 2011.

[76] K. Murugesen and J. Zhang. Hybrid hierarchical clustering: An experimental analysis. Technical report, University of Kentucky, 2011.

[77] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 511–524, 2014.

[78] S. Navathe, S. Ceri, G. Weiderhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Digital Library*, 9:680–710, 1984.

[79] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning of algorithms for database design. In *ACM Transactions on Database Systems*, pages 680–710, 1984.

[80] K. A. Nazeer and M. P. Sebastian. Improving the accuracy and efficiency of the k-means clustering algorithm. In *Proceedings of the World Congress on Engineering*, pages 263–304, July 2009.

[81] L. Ngan, M. Kirchberg, and R. Kanagasabai. Review of semantic web service discovery methods. *2010 IEEE Sixth World Congress On Services*, pages 176–177, June 2010.

[82] L. Nogueira, L. M. Pinho, and J. Coelho. Flexible and dynamic replication control for interdependent distributed real-time embedded systems. *IFIP Advances in Information and Communication Technology*, 329:66–77, 2010.

[83] M. T. Oszu and P. Valduriez. *Principles of Database Systems, Third Edition.* Pearson, 2011.

[84] A. Pavlo, C. Curino, and S. Zdonik. H-store. *Proceedings of the 2012 international conference on Management of Data*, pages 61–72, 2012.

[85] A. Pavlo, C. Curino, and S. Zdonik. Skew aware automatic database partitioning in shared nothing, parallel oltp systems. *Proceedings of the 2012 International Conference on Management of Data*, pages 61–72, 2012.

[86] S. Plantikow, K. Peter, M. Hogqvist, C. Grimme, and A. Papaspirou. Generalizing the data management of three community grids. *Future Generation of Computer Systems*, 25(3):281–289, March 2009.

[87] C. B. Pop, V. R. Chifu, I. Salomie, M. Dinsoreanu, T. David, and V. Acretoaie. Ant-inspired framework for automatic web service composition. *Scalable Computing: Practice and Experience*, 12(1):137–150, 2011.

[88] D. Pritchett. Base An acid alternative. *Object Relational Mapping*, 6:48–55, June 2008.

[89] F. Raab, W. Kohler, and A. Shah. Overview of the tpc benchmark c, the order entry benchmark. Technical report, Transaction Processing Performance Council, 2013.

[90] S. Rabah, D. Ni, P. Jahanshahi, and L. F. Guzman. Current state and challenges of automatic planning in web service composition. *Arxiv preprint arXiv:1107.1932, 2011*, pages 25–33, 2011.

[91] A. Rajasekar, R. Moore, and M. Wan. Storage resource broker: Managing distributed data in the grid. *Future Generation of Computer Systems*, 2006.

[92] P. Khrishna Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):154–169, 2004.

[93] L. Resende and R. Feng. Handling heterogeneous data sources in a soa environment with service data objects (sdo). *SIGMOD*, June 2007.

[94] L. Rodriguez and X. Li. Dynamic vertical partitioning of databases using active rules. *Database and Expert Systems Applications*, 7447:191–198, 2012.

[95] D. Sacca and G. Wiederhold. Database partitioning in a cluster of processors. In *ACM Transactions on Database Systems*, pages 29–56, March 1985.

[96] A. Silbershcatz, H. Korth, and S. Sudarshan. *Database System Concepts, Sixth Edition*. McGraw-Hill, 2010.

[97] J. Soares and N. Preguica. Database engines on multicores scale: A practical approach. *SAC*, April 2015.

[98] M. Stonebraker. Errors in database systems, eventual consistency, and the cap theorem. Technical report, Communications ACM, April 2010.

[99] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. OâĂŹNeil, P. OâĂŹNeil, A. Rasin, N. Tran, and S. Zdonik. Cstore-a column-oriented dbms. Technical report, MIT, 2006.

[100] O. V. Sukhoroslova, A. O. Rubtsov, and S. Y. Volkov. Development of distributed computing applications and services with everest cloud platform. *6th International Conference Distributed Computing and Grid Technologies in Science and Education*, July 2015.

[101] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8:1–9, 2014.

[102] P. N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison-Wesley, 2006.

[103] A. L. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. *IEEE 28th International Conference on Data Engineering*, pages 102–113, 2012.

[104] J. Tatemura, O. Po, and H. Hacigumus. Microsharding: A declarative approach to support elastic oltp workloads. *ACM SIGOPS*, 2012.

[105] E. Theocharopoulos and M. Jackson. *OGSA WS-DAIR 1.0, Version 1.0*. W3C, December 2008.

[106] A. Thomson, T. Diamond, S. C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2012.

[107] K. Tripp. Strategies for partitioning relational data warehouses in microsoft sql server. Technical report, IBM, 2005.

[108] K. Tripp. Toward open grid services architecture. Technical report, GLOBUS, 2012.

[109] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[110] P. Wang, Z. Ding, C. Jiang, and M. Zhou. Automatic web service composition based on uncertainty execution effects. *Services Computing, IEEE Transactions*, March 2013.

[111] J. S. Ward and A. Barker. A cloud computing survey: Developments and future trends in infrastructure as a service computing. *Cornell University Library*, pages 1–14, June 2013.

[112] Y. Wei, A. A. Aslinger, S. H. Son, and J. A. Stankovic. Order: A dynamic replication algorithm for periodic transactions in distributed real-time databases. *Proceedings of the Work-in-Progress Session of the 15th Euromicro Conference on Real-Time Systems*, 2003.

[113] D. H. Widyantoro, T. R. Ioerger, and J. Yen. An incremental approach to building a cluster hierarchy. In *IEEE International Conference on Data Mining*, pages 705–708, April 2002.

[114] R. Xu and D. Wunsch II. Survey of clustering methods. *IEEE Transactions on Neural Networks*, pages 645–678, May 2005.

[115] Q. Yu, L. Chen, and B. Li. Automatic web service composition based on uncertainty execution effects. *Elsevier, Computers and Electrical Engineering*, January 2015.

[116] E. Zahoor, O. Perrin, and C. Godart. Disc: A declarative framework for self-healing web services composition. *2010 IEEE International Conference on Web Services*, pages 25–33, 2010.