Cross-Tier Application & Data Partitioning of Web Applications for Hybrid Cloud Deployment

Nima Kaviani, Eric Wohlstadter, Rodger Lea

{{nkaviani, wohlstad}@cs, rlea@magic}.ubc.ca
University of British Columbia, Vancouver, Canada

Abstract. Hybrid cloud deployment offers flexibility in trade-offs between the cost-savings/scalability of the public cloud and control over data resources provided at a private premise. However, this flexibility comes at the expense of complexity in distributing a system over these two locations. For multi-tier web applications, this challenge manifests itself primarily in the partitioning of application- and database-tiers. While there is existing research that focuses on either application-tier or data-tier partitioning, we show that optimized partitioning of web applications benefits from both tiers being considered simultaneously. We present our research on a new cross-tier partitioning approach to help developers make effective trade-offs between performance and cost in a hybrid cloud deployment. In two case studies the approach results in up to 54% reduction in monetary costs compared to a premise only deployment and 56% improvement in execution time compared to a naïve partitioning where application-tier is deployed in the cloud and data-tier is on private infrastructure.

1 Introduction

While there are advantages to deploying Web applications on public cloud infrastructure, many companies wish to retain control over specific resources [8] by keeping them at a private premise. As a result, *hybrid cloud computing*, has become a popular architecture where systems are built to take advantage of both public and private infrastructure to meet different requirements. However, architecting an efficient distributed system across these locations requires significant effort. An effective partitioning should not only guarantee that privacy constraints and performance objectives are met, but also should deliver on one of the primary reasons for using the public cloud, a cheaper deployment.

In this paper we focus on partitioning of OLTP-style web applications. Such applications are an important target for hybrid architecture due to their popularity. Web applications follow the well known multi-tier architecture, generally consisting of tiers such as: client-tier, application-tier[1] (serving dynamic web content), and back-end data-tier. Since the hybrid architecture is motivated by the management of sensitive data resources, our research focuses on combined partitioning of the data-tier (which hosts data resources) and the application-tier (which directly uses data resources). Figure 1 shows a high-level diagram of

[1] In the rest of the paper we use the terms code and application-tier interchangeably.

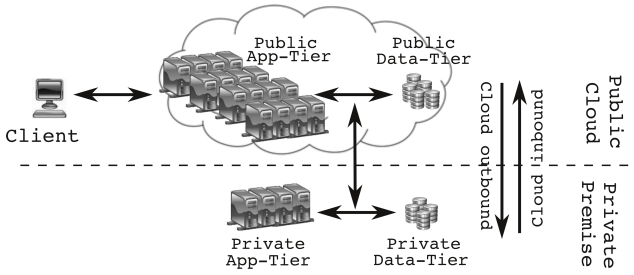


Fig. 1: High-level hybrid architecture with cross-tier partitioning of code and data.

these tiers being jointly partitioned across a hybrid architecture, which we refer to as *cross-tier partitioning*.

Existing research only applies partitioning to one of the application- or data tiers and does not address cross-tier partitioning. Systems such as CloneCloud [11], Cloudward Bound [14], Leymann et al.'s [20], and our own work on MANTICORE [17] partition only software but not data. Other work in the area provides for partitioning of relational databases [18] or Map-Reduce job/data components [6, 19, 29]. Unfortunately, one cannot "cobble together" a cross-tier solution by using independent results from such approaches. A new approach is needed that integrates application and data partitioning natively. Thus we argue that research into cross-tier partitioning is both *important* and *challenging*.

First, cross-tier partitioning is important because the data-flow between these tiers is tightly coupled. The application-tier can make several queries during its execution, passing information to and from different queries; an example is discussed in Section 2. Even though developers follow best practices to ensure the source code for the business logic and the data access layer are loosely coupled, this loose coupling does not apply to the data-flow. The data-flow crosscuts application- and data-tiers requiring an optimization that considers the two simultaneously. Any optimization must avoid, whenever possible, the latency and bandwidth requirements imposed by distributing such data-flow.

Second, cross-tier partitioning is challenging because it requires an analysis that simultaneously reasons about the execution of application-tier code and data-tier queries. On the one hand, previous work on partitioning of code is not applicable to database queries because it does not account for modeling of query execution plans. On the other hand, existing work on data partitioning does not account for the data-flow or execution footprint of the application-tier [18]. To capture a representation for cross-tier optimization, our contribution in this paper includes a new approach for modeling dependencies across both tiers as a combined *binary integer program* (BIP) [25].

We provide a tool which collects performance profiles of web application execution on a single host and converts it to the BIP format. The BIP is fed to an off-the-shelf optimizer whose output yields suggestions for placement of application- and data-tier components to either public cloud or private premise. Using proper tooling and middleware, a new system can now be distributed across the hybrid architecture using the optimized placement suggestions. To

the best of our knowledge, we provide the first approach for partitioning which integrates models of both application-tier and data-tier execution.

2 Motivating Scenario

As a motivating example, assume a company plans to take its on-premise trading software system and deploy it to a hybrid architecture. We use Apache DayTrader [1], a benchmark emulating the behavior of a stock trading system, to express this scenario. DayTrader implements business logic in the application-tier as different request types, for example, allowing users to login (`doLogin`), view/update their account information (`doAccount` & `doAccountUpdate`), etc. At the data-tier it consists of tables storing data for `account`, `accountprofile`, `holding`, `quote`, etc. Let us further assume that, as part of company regulations, user information (`account` & `accountprofile`) must remain on-premise.

Figure 2 shows the output of our cross-tier partitioning for `doLogin`. The figure shows the call-tree of function execution in the application-tier as well as data-tier query plans at the leaves. In the figure, we see four categories of components: (i) data on premise shown as black nodes, (ii) data in the cloud as square nodes, (iii) functions on premise as gray nodes, and (iv) functions in the cloud as white nodes. Here we use each of these four categories to motivate cross-tier partitioning.

First, some data is not suitable for deployment in the cloud due to privacy concerns or regulations [14]. Thus, many enterprises avoid committing deployment of certain data in the public cloud, instead hosting it on private premise infrastructure (e.g., `account` & `accountprofile` in Figure 2).

Second, function execution requires CPU resources which are generally cheaper and easier to scale in the public cloud (some reports claim a typical 80% savings using public cloud versus on-premise private systems [21]). Thus placing function execution in the public cloud is useful to limit the amount of on-premise infrastructure. So without regard to other factors, we would want to execute application-tier functions in the cloud.

Third, since we would like to deploy functions to the cloud, the associated data bound to those functions should be deployed to the cloud, otherwise we will incur additional latency and bandwidth usage. So there is motivation to

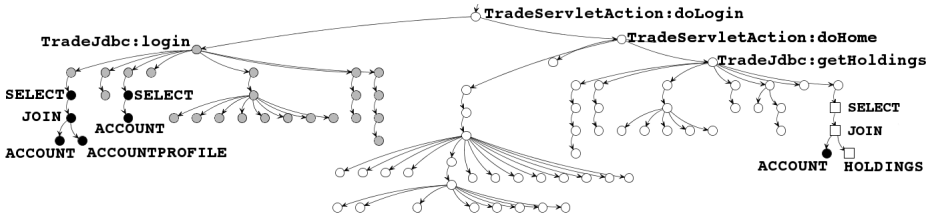


Fig. 2: A cross-tier partitioning suggested by our tool for the `doLogin` request from DayTrader showing a partitioned application- and data-tier: data on premise (black nodes), data in the cloud (square nodes), functions on premise (gray nodes), and functions in the cloud (white nodes).

move non-sensitive data to the cloud. However, such non-sensitive data may be bound to sensitive data through queries which operate over both. For this reason, moving non-sensitive data to the public cloud is not always a winning proposition. We will need an analysis which can reason about the benefit of moving data closer to functions executing in the public cloud versus the drawback of pulling it away from the sensitive data on premise.

Finally, it turns out that executing all functions in the public cloud is also not always a winning proposition. Some functions are written as transactions over several data resources. Such functions may incur too much communication overhead if they execute in the public cloud but operate on private premise data. So the benefit of executing them in the cloud needs to be balanced with this overhead.

These four cases help to illustrate the inter-dependencies between the application-tier and data-tier. In the case of `doLogin`, a developer may manually arrive at a similar partitioning with only minor inconvenience. However, to cover an entire application, developers need to simultaneously reason about the effects of component placements across all request types. This motivates the need for research on automation for cross-tier partitioning.

3 Background: Application-Tier Partitioning

Binary Integer Programming [25] has been utilized previously for partitioning of applications (although not for cross-tier partitioning) [10, 17, 22, 30]. A binary integer program (BIP) consists of the following:

- Binary variables: A set of binary variables $x_1, x_2, ..., x_n \in \{0, 1\}$.
- Constraints: A set of linear constraints between variables where each constraint has the form: $c_0x_0+c_1x_1+...+c_nx_n \{\leq, =, \geq\} c_m$ and c_i is a constant.
- Objective: A linear expression to minimize or maximize: $cost_1x_1+cost_2x_2+ ... + cost_nx_n$, with $cost_i$ being the cost charged to the model when $x_i = 1$. The job of a BIP optimizer is to choose the set of values for the binary variables which minimize/maximize this expression.

Formulating a cross-tier BIP for partitioning will require combining one BIP for the application-tier and another for the data-tier. Creating each BIP consists of the same high-level steps (although the specific details vary): (i) profiling, (ii) analysis, (iii) generating the BIP constraints and (iv) generating the BIP objective function. The overall process of applying cross-tier partitioning is shown in Figure 3. In the top left we see an application before partitioning. Notice that the profiling results are split in two branches. Here we focus on the flow following from the Profiling Logs branch, discussing the Explain Plan flow in Section 4. Our approach for generating a BIP for the application-tier follows from our previous work on Manticore [17] and is summarized as background here.

Profiling: The typical profiling process for application partitioning starts by taking existing software and applying instrumentation of its binaries. The software is then exercised on representative workloads, using the instrumentation to collect data on measured CPU usage of software functions and data exchange between them. This log of profiling information will be converted to the relevant variables and costs of a BIP.

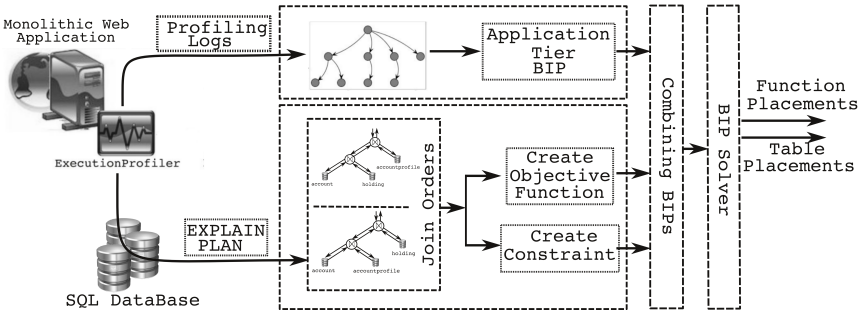


Fig. 3: The overall process of applying cross-tier partitioning to a monolithic web application (process flows from left to right).

Analysis: The log of profile data is converted to a graph model before being converted to a BIP, as shown in the top flow of Figure 3. Let $App(V, E)$ represent a model of the application, where $\forall v \in V$, v corresponds to a function execution in the application. Similarly $\forall u, v \in V$, $e_{(u,v)} \in E$ implies that there is data exchange between functions in the application corresponding to u and v in *App*. $\forall e_{(u,v)} \in E$, we define $d_{u \leftrightarrow v}$ as the amount of data exchanged between u to v.

BIP Constraints: The graph model is then used to formulate a BIP. For every node u in the model we consider a variable $x_u \in \{0, 1\}$. Using input from a developer some nodes can be constrained to a particular location by fixing their value, e.g., (0: private premise, 1: public cloud). Unconstrained variables are free for an optimizer to choose their values so as to minimize the objective function. These values are then translated to placement decisions for function executions.

BIP Objective: For each $v \in V$ we define $cost_{exec_v}$ to represent the cost of executing v on-premise and $cost'_{exec_v}$ to represent cost of executing v in the cloud. We also define $latency_{(u,v)}$ to represent the latency cost on edge $e_{(u,v)}$ and calculate the communication cost ($cost_{comm_{u,v}}$) for edge $e_{(u,v)}$ as follows:

$$cost_{comm_{u,v}} = latency_{(u,v)} + \frac{d_{u \leftrightarrow v}}{D_{unit}} \times cost_{comm_{unit}} \tag{1}$$

where D_{unit} would be the unit of data to which cloud data charges are applied and $cost_{comm_{unit}}$ would be the cloud charges for D_{unit} of data transfer, and $d_{u \leftrightarrow v}$ represents data exchange between vertices u and v. As demonstrated by work such as Cloudward Bound [14], in a cloud computing setting such raw performance costs such as measured CPU usage and data transfer can be converted to monetary costs using the advertised infrastructure costs of vendors such as Amazon EC2. This allows developers to optimize for trade-offs in performance cost and monetary cost objectives.

Using such costs we can define an objective expression (The non-linear expression in the objective function can be relaxed by making the expansion in [22]):

$$\min \sum_{i \in V} x_i cost_{exec_i} + \sum_{(i,j) \in E} (x_i - x_j)^2 cost_{comm_{i,j}} \tag{2}$$

Finally, the BIP is fed to a solver which determines an assignment of functions to locations. By choosing the location for each function execution, the optimizer chooses an efficient partitioning by placing functions in the cloud when possible if it does not introduce too much additional latency or bandwidth requirements.

Different from previous work, our cross-tier partitioning incorporates a new BIP model of query plan execution into this overall process. In the next section, we describe these details which follow the bottom flow of Figure 3.

4 BIP for Data-Tier Partitioning

The technical details of extending application-tier partitioning to integrate the data-tier are motivated by four requirements: (i) weighing the benefits of distributing queries, (ii) comparing the trade-offs between join orders, (iii) taking into account intra-request data-dependencies and (iv) providing a query execution model comparable to application-tier function execution. In this section, we first further motivate cross-tier partitioning by describing each of these points, then we cover the technical details for the steps of partitioning as they relate to the data-tier. We focus on a data-tier implemented with a traditional SQL database. While some web application workloads can benefit from the use of alternative NoSQL techniques, we chose to focus initially on SQL due to its generality and widespread adoption.

First, as described in Section 2, placing more of the less-sensitive data in the cloud will allow for the corresponding code from the application-tier to also be placed in the cloud, thus increasing the overall efficiency of the deployment and reducing data transfer. However, this can result in splitting the set of tables used in a query across public and private locations. For our DayTrader example, each user can have many stocks in her holdings which makes the HOLDING table quite large. As shown in Figure 2, splitting the *join* operation can push the HOLDINGS table to the cloud (square nodes) and eliminate the traffic of moving its data to the cloud. This splitting also maintains our constraint to have the privacy sensitive ACCOUNT table on the private premise. An effective modeling of the data-tier needs to help the BIP optimizer reason about the trade-offs of distributing such queries across the hybrid architecture.

Second, the order that tables are joined can have an effect not only on traditional processing time but also on round-trip latency. We use a running example throughout this section of the query shown in Figure 4, with two different join orders, left and right. If the query results are processed in the public cloud where the HOLDING table is in the cloud and ACCOUNT and ACCOUNTPROFILE are stored on the private premise, then the plan on the left will incur two-round trips from the public to private locations for distributed processing. On the other hand, the query on the right only requires one round-trip. Modeling the data-tier should help the BIP optimizer reason about the cost of execution plans for different placements of tables.

Third, some application requests execute more than one query. In these cases, it may be beneficial to partition functions to group execution with data at a single location. Such grouping helps to eliminate latency overhead otherwise needed to move data to the location where the application-tier code executes.

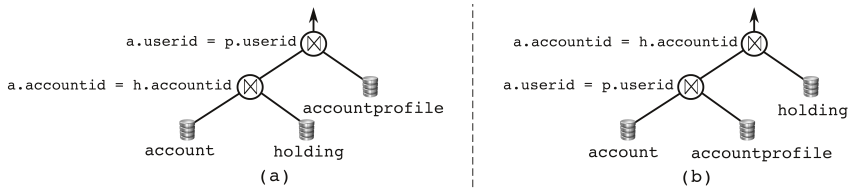


Fig. 4: Two possible query plans from one of the queries in DayTrader:
`SELECT p.*, h.* FROM holding h, accountprofile p, account a WHERE h.accountid = a.accountid AND a.userid = p.userid AND h.quote_symbol = ? AND a.accountid = ?`

An example of this is shown in Figure 2, where a sub-tree of function executions for `TradeJdbc:login` are labeled as "private" (gray nodes). By pushing this sub-tree to the private premise, the computation needed for working over ACCOUNT and ACCOUNTPROFILE data in the two queries under `TradeJdbc:login` can be completed at the premise without multiple round-trips between locations.

Fourth, since the trade-offs on function placement depend on the placement of data and vice-versa, we need a model that can reason simultaneously about both application-tier function execution and query plan execution. Thus the model for the data-tier should be compatible for integration with an approach to application partitioning such as the one described in Section 3.

Having motivated the need for a model of query execution to incorporate the data-tier in a cross-tier partitioning, we now explore the details, following the bottom flow of Figure 3. The overall process is as follows. We first, profile query execution using EXPLAIN PLAN (Section 4.1). This information is used to collect statistics for query plan operators by interrogating the database for different join orders (Section 4.2). The statistics are then used to generate both BIP constraints (Section 4.3) and a BIP objective function (Section 4.4). Finally, these constraints and objective are combined with that from the application-tier to encode a cross-tier partitioning model for a BIP solver.

4.1 Database Profiling with EXPLAIN PLAN

Profiling information is available for query execution through the EXPLAIN PLAN SQL command. Given a particular query, this command provides a tree-structured result set detailing the execution of the query. We use a custom JDBC driver wrapper to collect information on the execution of queries. During application profiling (cf. Section 3) whenever a query is issued by the application-tier, our JDBC wrapper intercepts the query and collects the plan for its execution. The plan returned by the database contains the following information:

1. `type`(op): Each node in the query plan is an operator such as a join, table access, selection (i.e. filter), sort, etc. To simplify presentation of the technical details, we assume that each operator is either a join or a table access. Other operators are handled by our implementation but they don't add extra complexity compared to a join operator. For example, in Figure 4, the selection (i.e. filter) operators are elided. We leverage the database's own cost model directly by recording from the provided plan how much each operator

costs. Hence, we don't need to evaluate different operator implementations to evaluate their costs. On the other hand, we do need to handle joins specially because table placement is greatly affected by their ordering.

2. `cpu`(*op*): This statistic gives the expected time of execution for a specific operator. In general, we assume that the execution of a request in a hybrid web application will be dominated by the CPU processing of the application-tier and the network latency. So in many cases, this statistic is negligible. However, we include it to detect the odd case of expensive query operations which can benefit from executing on the public cloud.
3. `size`(*op*): This statistic captures the expected number of bytes output by an operator which is equal to the expected number of rows times the size of each retrieved row. From the perspective of the plan tree-structure, this is the data which flows from a child operator to its parent.
4. `predicates`(*joinOp*): Each join operator combines two inputs based on a set of predicates which relate those inputs. We use these predicates to determine if alternative join orders are possible for a query.

When profiling the application, the profiler observes and collects execution statistics only for plans that get executed but not for alternative join orders. However, the optimal plan executed by the database engine in a distributed hybrid deployment can be different from the one observed during profiling. In order to make the BIP partitioner aware of alternative orders, we have extended our JDBC wrapper to consult the database engine and examine the alternatives by utilizing a combination of Explain Plan and join order hints. Our motivation is to leverage the already existing cost model from a production database for cost estimation of local operator processing, while still covering the space of all query plans. The profiler also captures which sets of tables are accessed together as part of an atomic transaction. This information is used to model additional costs of applying a two-phase commit protocol, should the tables get partitioned.

4.2 Join Order Enumeration

We need to encode enough information in the BIP so it can reason over all possible plans. Otherwise, the BIP optimizer would mistakenly assume that the plan executed during our initial profiling is the only one possible. For example, during initial profiling on a single host, we may only observe the left plan from Figure 4. However, in the example scenario, we saw that the right plan introduces fewer round-trips across a hybrid architecture. We need to make sure the right plan is accounted for when deciding about table placement. Our strategy to collect the necessary information for all plans consists of two steps: (i) gather statistics for all operators in all plans irrespective of how they are joined, and (ii) encode BIP constraints about how the operators from step (i) can be joined. Here we describe step 1 and then describe step 2 in the next subsection. The novelty of our approach is that instead of optimizing to a specific join order in isolation of the structure of application-tier execution, we encode the possible orders together with the BIP of the application-tier as a combined BIP.

As is commonly the case in production databases, we assume a query plan to be left-deep. In a left-deep query plan, a join takes two inputs: one from a

single base relation (i.e. table) providing immediate input (referred to as the "inner relation"); and another one potentially derived as an intermediate result from a different set of relations (the "outer relation"). The identity of the inner relation and the set of tables comprising the outer relation uniquely determine the estimated best cost for an individual join operator. This is true regardless of the in which the outer relation was derived [26]. For convenience in our presentation, we call this information the operator's *id*, because we use it to represent an operator in the BIP. For example, the root operator in Figure 4a takes ACCOUNTPROFILE as an inner input and {HOLDING, ACCOUNT} as an outer input. The operator's id is then {(HOLDING, ACCOUNT), ACCOUNTPROFILE}. We will refer to the union of these two inputs as a *join set* (the set of tables joined by that operator). For example, the join set of the aforementioned operator is {HOLDING, ACCOUNT, ACCOUNTPROFILE}. Notably, while the join sets for the roots of Figures 4a & 4b are the same, Figures 4b's root node has the operator id {(ACCOUNTPROFILE, ACCOUNT), HOLDING} allowing us to differentiate the operators in our BIP formulation. Our task in this section is to collect statistics for the possible join operators with unique ids.

Most databases provide the capability for developers to provide hints to the query optimizer in order to force certain joins. For example in Oracle, a developer can use the hint `LEADING(X, Y, Z, ...)`. This tells the optimizer to create a plan where X and Y are joined first, then their intermediate result is joined with Z, etc. We use this capability to extract statistics for all join orders.

Algorithm 1 takes as input a query observed during profiling. In line 2, we extract the set of all tables referenced in the query. Next, we start collecting operator statistics for joins over two tables and progressively expand the size through each iteration of the loop on line 3. The table t, selected for each iteration of line 4 can be considered as the inner input of a join. Then, on line 5 we loop through all sets of tables of size i which don't contain t. On line 6, we verify if t is joinable with the set S by making sure that at least one table in the set S shares a join (access) predicate with t. This set forms the outer input to a join. Finally, on line 7, we collect statistics for this join operator by forcing the database to explain a plan in which the join order is prefixed by the outer input set, followed by the inner input relation. We record the information for each operator by associating it with its id. For example, consider Figure 4 as the input Q to Algorithm 1. In a particular iteration of line 5, i might be chosen as

```
1 Function collectOperatorStats(Q)
2 |   tables ← getTables(Q);
3 |   for i ← 1 to |tables| do
4 |   |   foreach t ∈ tables do
5 |   |   |   foreach S ∈ P_i(tables − {t}) do
6 |   |   |   |   if isJoinable(S, t) then
7 |   |   |   |   |   explainPlanWithLeadingRelations(S, t);
```

Algorithm 1: Function to collect statistics for alternative query plan operators for the input query Q. $\mathcal{P}_i$ is the powerset operator over sets of size i.

2 and t as ACCOUNTPROFILE. Since ACCOUNTPROFILE has a predicate shared with ACCOUNT, S could be chosen as the set of size 2: {ACCOUNT, HOLDINGS}. Now on line 6, `explainPlanWithLeadingTables`({ACCOUNT, HOLDINGS}, ACCOUNTPROFILE) will get called and the statistics for the join operator with the corresponding id will get recorded.

The bottom-up structure of the algorithm follows similarly to the classic dynamic programming algorithm for query optimization [26]. However, in our case we make calls into the database to extract costs by leveraging EXPLAIN PLAN and the `LEADING` hint. The complexity of Algorithm 1 is $O(2^n)$ (where n is the number of tables); which is the same as the classic algorithm for query optimization [26], so our approach scales in a similar fashion. Even though this is exponential, OLTP queries typically don't operate on over more than ten tables.

4.3 BIP Constraints

Now that we know the statistics for all operators with a unique id, we need to instruct the BIP how they can be composed. Our general strategy is to model each query plan operator, *op*, as a binary variable in a BIP. The variable will take on the value 1 if the operator is part of the query plan which minimizes the objective of the BIP and 0 otherwise. Each possible join set is also modeled as a variable. Constraints are used to create a connection between operators that create a join set and operators that consume a join set (cf. Table 1). The optimizer will choose a plan having the least cost given both the optimizers choice of table placement and function execution placement (for the application-tier). Each operator also has associated variables op_{cloud} and $op_{premise}$ which indicate

Function	`genChoice`($joinSet$, {op_1 ... op_n})
Generated constraint	$op_1 + ... + op_n = joinSet$
Description	a $joinSet$ is produced by one and only one of the operators op_1 ... op_n

Function	`genInputConstraint`(op, {in_{left}, in_{right}})
Generated constraint	$-2 \times op + in_{left} + in_{right} \geq 0$
Description	If op is 1, then variables representing its left and right inputs (in_{left} and in_{right}) must both be 1

Table 1: Constraint generation functions

```
1 Function createConstraints(joinSet)
2     ops ← getOperatorsForJoinSet(joinSet);
3     genChoice(joinSet, ops);
4     foreach op ∈ ops do
5         inputs ← getInputs(op);
6         genInputConstraint(op, inputs);
7         if sizeof(left(inputs)) > 0 then
8             createConstraints(left(inputs));
```

Algorithm 2: Constraint generation, using functions from Table 1. The details for the functions `getOperatorsForJoinSet`, `getInputs`, `sizeof`, and `left` are not shown but their uses are described in the text.

the placement of the operator. Table placement is controlled by each table's associated table access operators. The values of these variables for operators in the same query plan will allow us to model the communication costs associated with distributed queries.

Our algorithm to formulate these composition constraints makes use of two helper functions as shown in Table 1, namely `genChoice` and `genInputConstraint`. When these functions are called by our algorithms, they append the generated constraint to the BIP that was already built for the application-tier. The first function, `genChoice`, encodes that a particular join set may be derived by multiple possible join operators (e.g., {HOLDING, ACCOUNT, ACCOUNTPROFILE} could be derived by either of the root nodes in Figure 4). The second function, `genInputConstraint`, encodes that a particular join operator takes as inputs the join sets of its two children. It ensures that if *op* is selected, both its children's join sets (in_{left} and in_{right}) are selected as well, constraining which subtrees of the execution plan can appear under this operator. The "$\geq$" inequality in Table 1 helps to encode the boolean logic $op \rightarrow in_{left} \wedge in_{right}$.

Starting with the final output join set of a query, Algorithm 2 recursively generates these constraints encoding choices between join operators and how parent operators are connected to their children. It starts on line 2 by calling a function to retrieve all operator ids which could produce that join set (these operators were all collected during the execution of Algorithm 1). It passes this information to `genChoice` on line 3. On line 4, we loop over all these operator ids, decomposing each into its two inputs on line 5. This information is then passed to `genInputConstraint`. Finally on line 7, we test for the base case of a table access operator. If we have not hit the base case, then the left input becomes the join set for recursion on line 8.

4.4 BIP Objective

Creating the optimization objective function consists of two parts: (i) determining the costs associated with the execution of individual operators, and (ii) creating a mathematical formulation of those costs. The magnitude of the execution cost for each operator and the communication cost between operators that are split across the network are computed using a similar cost model to previous work [31]. This accounts for the variation between local execution and distributed execution in that the latter will make use of a semi-join optimization to reduce costs (i.e. input data to a distributed join operator will transmit only the columns needed to collect matching rows). We extend the previous cost model to account for possible transaction delays. We assume that if the tables involved in an atomic transaction are split across the cloud and the private premise, by default the transaction will be resolved using the two-phase commit protocol.

Performance overhead from atomic two-phase distributed transactions comes primarily from two sources: protocol overhead and lock contention. Protocol overhead is caused by the latency of prepare and commit messages in a database's two-phase commit protocol. Lock contention is caused by queuing delay which increases as transactions over common table rows become blocked. We provide two alternatives to account for such overhead:

Function	`genAtMostOneLocation`(op)
Generated constraint	$op_{cloud} + op_{premise} = op$
Description	If the variable representing op is 1, then either the variable representing it being placed in the cloud is 1 or the variable representing it being place in the premise is 1

Function	`genSeparated`(op_1, op_2)
Generated constraint	$op_{1cloud} + op_{2premise}$ - $cut_{op_1,op_2} \leq 1$ $op_{1premise} + op_{2cloud}$ - $cut_{op_1,op_2} \leq 1$
Description	If the variables representing the locations of two operators are different, then the variable cut_{op_1,op_2} is 1

Table 2: Functions for generating objective helper constraints

- For some transactions, lock contention is negligible. This is because the application semantics don't induce sharing of table rows between multiple user sessions. For example, in DayTrader, although ACCOUNT and HOLDINGS tables are involved in an atomic transaction, specific rows of these tables are only ever accessed by a single user concurrently. In such cases we charge the cost of two extra round-trips between the cloud and the private premise to the objective function, one to prepare the remote site for the transaction and another to commit it.
- For cases where lock contention is expected to be considerable, developers can request that certain tables be co-located in any partitioning suggested by our tool. This prevents locking for transactions over those tables to be delayed by network latency. Since such decisions require knowledge of application semantics that are difficult to infer automatically, our tool provides an interactive visualization of partitioning results, as shown in Figure 2. This allows developers to work through different "what-if" scenarios of table co-location constraints and the resulting suggested partitioning.

Next, we need to encode information on CPU and data transmission costs into the objective function. In addition to generating a BIP objective, we will need some additional constraints that ensure the calculated objective is actually feasible. Table 2 shows functions to generate these constraints. The first constraint specifies that if an operator is included as part of a chosen query plan (its associated id variable is set to 1), then either the auxiliary variable op_{cloud} or $op_{premise}$ will have to be 1 but not both. This enforces a single placement location for op. The second builds on the first and toggles the auxiliary variable cut_{op_1,op_2} when op_{1cloud} and $op_{2premise}$ are 1, or when $op_{1premise}$ and op_{2cloud} are 1.

The objective function itself is generated using two functions in Table 3. The first possibly charges to the objective function either the execution cost of the operator on the cloud infrastructure or on the premise infrastructure. Note that it will never charge both due to the constraints of Table 2. The second function charges the communication cost between two operators if the associated *cut* variable was set to 1. In the case that there is no communication between two operators this cost is simply 0.

Algorithm 3 takes a join set as input and follows a similar structure to Algorithm 2. The outer loop on line 3, iterates over each operator that could produce the particular join set. It generates the location constraints on line 4 and the

execution cost component to the objective function on line 5. Next, on line 7, it iterates over the two inputs to the operator. For each, it extracts the operators that could produce that input (line 8) and generates the communication constraint and objective function component. Finally, if the left input is not a base relation (line 11), it recurses using the left input now as the next join set.

Having appended the constraints and objective components associated with query execution to the application-tier BIP, we make a connection between the two by encoding the dependency between each function that executes a query and the possible root operators for the associated query plan.

5 Implementation

We have implemented our cross-tier partitioning as a framework. It conducts profiling, partitioning, and distribution of web applications which have their business logic implemented in Java. Besides the profiling data, the analyzer also accepts a declarative XML policy and cost parameters. The cost parameters encode the monetary costs charged by a chosen cloud infrastructure provider and expected environmental parameters such as available bandwidth and network latency. The declarative policy allows for specification of database table placement and co-location constraints. In general we consider the placement of privacy sensitive data to be the primary consideration for partitioning decisions.

Function	`genExecutionCost`(op)
Generated objective component	$op_{cloud} \times$ `execCost`$_{cloud}$`(`op`)` + $op_{premise} \times$ `execCost`$_{premise}$`(`op`)`
Description	If the variable representing op deployed in the cloud/premise is 1, then charge the associated cost of executing it in the cloud/premise respectively

Function	`genCommCost`(op_1, op_2)
Generated objective component	$cut_{op_1,op_2} \times$ `commCost`(op_1, op_2)
Description	If cut_{op_1,op_2} for two operators op_1 and op_2 was set to 1, then charge their cost of communication

Table 3: Functions for generating objective function

```
1  Function createObjFunction(joinSet)
2      ops ← getOperatorsForJoinSet(joinSet);
3      foreach op ∈ ops do
4          genAtMostOneLocation(op);
5          genExecutionCost(op);
6          inputs ← getInputs(op);
7          foreach input ∈ inputs do
8              foreach childOp ∈ getOperatorsForJoinSet(input) do
9                  genSeparated(op, childOp);
10                 genCommCost(op, childOp);
11         if sizeof(left(inputs)) > 0 then
12             createObjFunction(left(inputs));
```

Algorithm 3: Objective generation

However, developers may wish to monitor and constrain the placement of function executions that operate over this sensitive data. For this purpose we rely on existing work using taint tracking [9] which we have integrated into our profiler.

For partitioning, we use the off-the-shelf integer programming solver `lp_solve` [2] to solve the discussed BIP optimization problem. The results lead to generating a *distribution plan* describing which entities need to be separated from one another (*cut-points*). A cut-point may separate functions from one another, functions from data, and data from one another. Separation of code and data is achievable by accessing the database engine through the database driver. Separating inter-code or inter-data dependencies requires extra middleware.

For functions, we have developed a bytecode rewriting engine as well as an HTTP remoting library that takes the partitioning plan generated by the analyzer, injects remoting code at each *cut-point*, and serializes data between the two locations. This remoting instrumentation is essentially a simplified version of J-Orchestra [28] implemented over HTTP (but is not yet as complete as the original J-Orchestra work). In order to allow for distribution of data entities, we have taken advantage of Oracle's distributed database management system (DDBMS). This allows for tables remote to a local Oracle DBMS, to be identified and queried for data through the local Oracle DBMS. This is possible by providing a database link (`@dblink`) between the local and the remote DBMS systems. Once a bidirectional `dblink` is established, the two databases can execute SQL statements targeting tables from one another. This allows us to use the distribution plan from our analyzer system to perform vertical sharding at the level of database tables. Note that the distributed query engine acts on the deployment of a system after a decision about the placement of tables has been made by our partitioning algorithm. We have provided an Eclipse plugin implementation of the analyzer framework available online [3].

6 Evaluation

We evaluate our work using two different applications: *DayTrader* [1] and *RUBiS* [4]. DayTrader (cf. Section 2) is a Java benchmark of a stock trading system. RUBiS implements the functionality of an auctioning Web site. Both applications have already been used in evaluating previous cloud computing research [16, 27].

We can have 9 possible deployment variations with each of the data-tier and the application tier being (i) on the private premise, (ii) on the public cloud, or (iii) partitioned for hybrid deployment. Out of all the placements we eliminate the 3 that place all data in the cloud as it contradicts the constraints to have privacy sensitive information on-premise. Also, we consider deployments with only data partitioned as a subset of deployments with both code and data partitioned, and thus do not provide separate deployments for them. The remaining four models deployed for evaluations were as follows: (i) both code and data are deployed to the premise (*Private-Premise*); (ii) data is on-premise and code is in the cloud (*Naïve-Hybrid*); (iii) data is on-premise and code is partitioned (*Split-Code*); and (iv) both data and code are partitioned (*Cross-Tier*).

For both DayTrader and RUBiS, we consider privacy incentives to be the reason behind constraining placement for some database tables. As such, when

partitioning data, we constrain tables storing user information (`account` and `accountprofile` for DayTrader and `users` for RUBiS) to be placed on-premise. The remaining tables are allowed to be flexibly placed on-premise or in the cloud.

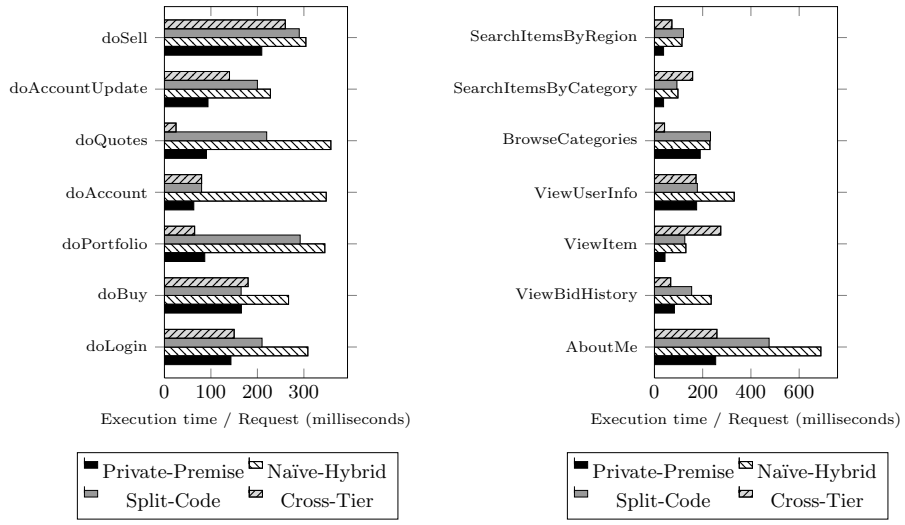
We used the following setup for the evaluation: for the premise machines, we used two 3.5 GHz dual core machines with 8.0 GB of memory, one as the application server and another as our database server. Both machines were located at our lab in Vancouver, and were connected through a 100 Mb/sec data link. For the cloud machines, we used an extra large EC2 instance with 8 EC2 Compute Units and 7.0 GB of memory as our application server and another extra large instance as our database server. Both machines were leased from Amazon's US West region (Oregon) and were connected by a 1 Gb/sec data link. We use Jetty as the Web server and Oracle 11g Express Edition as the database servers. We measured the round-trip latency between the cloud and our lab to be 15 milliseconds. Our intentions for choosing these setups is to create an environment where the cloud offers the faster and more scalable environment. To generate load for the deployments, we launched simulated clients from a 3.0 GHz quad core machine with 8 GB of memory located in our lab in Vancouver. In the rest of this section we provide the following evaluation results for the four deployments described above: execution times (Section 6.1), expected monetary deployment costs (Section 6.2), and scalability under varying load (Section 6.3).

6.1 Evaluation of Performance

We measured the execution time across all business logic functionality in DayTrader and RUBiS under a load of 100 requests per second, for ten minutes. By execution time we mean the elapsed wall clock time from the beginning to the end of each servlet execution. Figure 5 shows those with largest average execution times. We model a situation where CPU resources are not under significant load. As shown in Figure 5, execution time in cross-tier partitioning is significantly better than any other model of hybrid deployment and is closely comparable to a non-distributed private premise deployment. As an example, execution time for DayTrader's `doLogin` under *Cross-Tier* deployment is 50% faster than *Naïve-Hybrid* while `doLogin`'s time for *Cross-Tier* is only 5% slower compared to *Private-Premise* (i.e., the lowest bar in the graph). It can also be seen that, for `doLogin`, *Cross-Tier* has 25% better execution time compared to *Split-Code*, showing the effectiveness of cross-tier partitioning compared to partitioning only at the application-tier.

Similarly for other business logic functionality, we note that cross-tier partitioning achieves considerable performance improvements when compared to other distributed deployment models. It results in performance measures broadly similar to a full premise deployment. For the case of DayTrader - across all business logic functionality of Figure 5a - *Cross-Tier* results in an overall performance improvement of 56% compared to *Naïve-Hybrid* and a performance improvement of around 45% compared to *Split-Code*.

We observed similar performance improvements for RUBiS. *Cross-Tier* RUBiS performs 28.3% better - across all business logic functionality of Figure 5b - compared to its *Naïve-Hybrid*, and 15.2% better compared to *Split-Code*. Based



(a) Execution times for DayTrader (b) Execution times for RUBiS

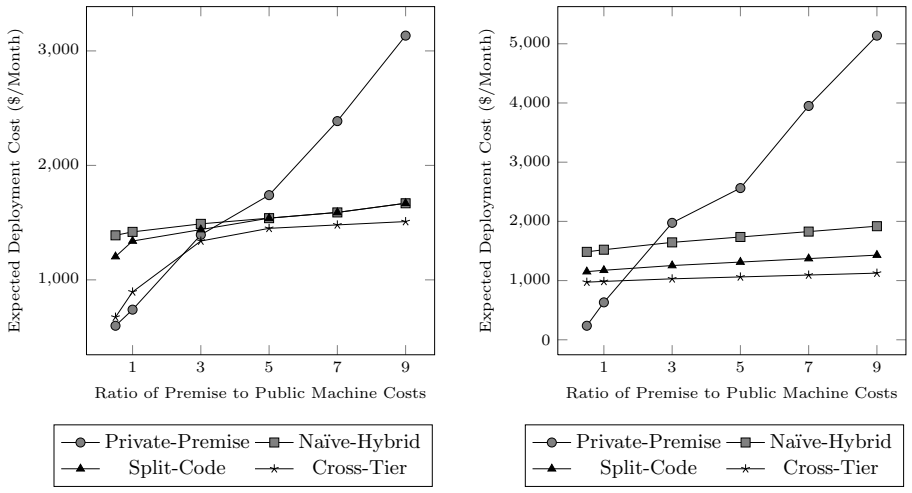
Fig. 5: Measured execution times for selected request types in the four deployments of DayTrader and RUBiS.

on the results, cross-tier partitioning provides more flexibility for moving function execution to the cloud and can significantly increase performance for a hybrid deployment of an application.

6.2 Evaluation of Deployment Costs

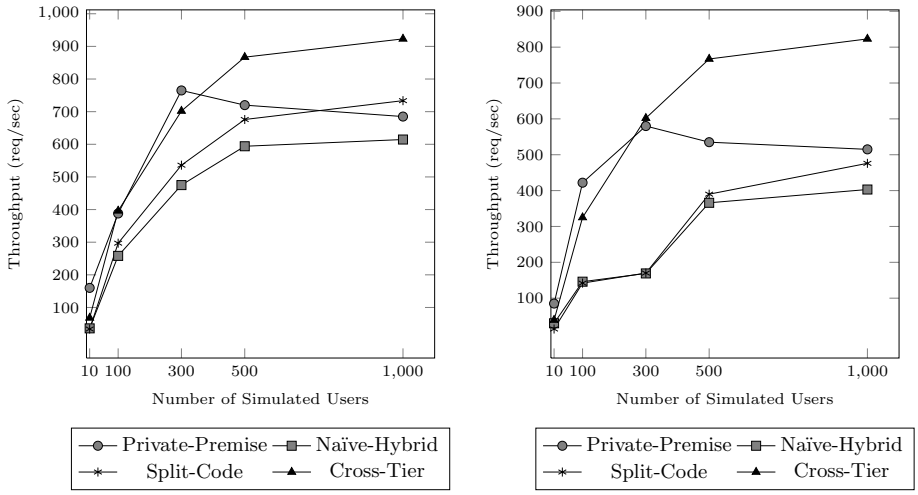
For computing monetary costs of deployments, we use parameters taken from the advertised Amazon EC2 service where the cost of an extra large EC2 instance is \$0.48/hour and the cost of data transfer is \$0.12/GB. To evaluate deployment costs, we apply these machine and data transfer costs to the performance results from Section 6.1, scale the ten minute deployment times to one month, and gradually change the ratio of premise-to-cloud deployment costs to assess the effects of varying cost of private premise on the overall deployment costs.

As shown in both graphs, a *Private-Premise* deployment of web applications results in rapid cost increases, rendering such deployments inefficient. In contrast, all partitioned deployments of the applications result in more optimal deployments with *Cross-Tier* being the most efficient. For a cloud cost 80% cheaper than the private-premise cost (5 times ratio), DayTrader's *Cross-Tier* is 20.4% cheaper than *Private-Premise* and 11.8% cheaper than *Naïve-Hybrid* and *Split-Code* deployments. RUBiS achieves even better cost savings with *Cross-Tier* being 54% cheaper than *Private-Premise* and 29% cheaper than *Naïve-Hybrid* and *Split-Code*. As shown in Figure 6a, in cases where only code is partitioned, a gradual increase in costs for machines on-premise eventually results in the algorithm pushing more code to the cloud to the point where all code is in the cloud and all data is on-premise. In such a situation *Split-Code* eventually converges to



(a) Monthly deployment costs for DayTrader (b) Monthly deployment costs for RUBiS

Fig. 6: Monthly cost comparison for different deployments of DayTrader and RUBiS.



(a) Scalability tests for DayTrader (b) Scalability tests for RUBiS

Fig. 7: Scalability tests for full premise, full cloud, and hybrid deployments

Naïve-Hybrid; i.e., pushing all the code to the cloud. Similarly, *Cross-Tier* will finally stabilize. However since in *Cross-Tier* part of the data is also moved to the cloud, the overall cost is lower than *Naïve-Hybrid* and *Split-Code*.

6.3 Evaluation of Scalability

We also performed scalability analyses for both DayTrader and RUBiS to see how different placement choices affect application throughput. DayTrader comes

with a random client workload generator that dispatches requests to all available functionality on DayTrader. On the other hand, RUBiS has a client simulator designed to operate either in the browsing mode or the buy mode.

For both DayTrader and RUBiS we used a range of 10 to 1000 client threads to send requests to the applications in 5 minute intervals with 1 minute ramp-up. For RUBiS, we used the client in buy mode. Results are shown in Figure 7. As the figure shows, for both applications, after the number of requests reaches a certain threshold, *Private-Premise* becomes overloaded. For *Naïve-Hybrid* and *Split-Code*, the applications progressively provide better throughput. However, due to the significant bottleneck when accessing the data, both deployments maintain a consistent but rather low throughput during their executions. Finally, *Cross-Tier* achieved the best scalability. With a big portion of the data in the cloud, the underlying resources for both code and data can scale to reach a much better overall throughput for the applications. Despite having part of the data on the private premise, due to its small size the database machine on premise gets congested at a slower rate and the deployment can keep a high throughput.

7 Related Work

Our research bridges the two areas of application and database partitioning but differs from previous work in that it uses a new BIP formulation that considers both areas. Our focus is not on providing all of the many features provided by every previous project either on application partitioning or database partitioning. Instead, we have focused on providing a new interface between the two using our combined BIP. We describe the differences in more detail by first describing some related work in application partitioning and then database partitioning.

Application Partitioning: Coign [15] is an example of classic application partitioning research which provides partitioning of Microsoft COM components. Other work focuses specifically on partitioning of web/mobile applications such as Swift [10], Hilda [30], and AlfredO [24]. However that work is focused on partitioning the application-tier in order to off-load computation from the server-side to a client. That work does not handle partitioning of the data-tier.

Minimizing cost and improving performance for deployment of software services has also been the focus of cloud computing research [19]. While approaches like Volley [6] reduce network traffic by relocating data, others like CloneCloud [11], Cloudward Bound[14], and our own Manticore [17] improve performance through relocation of server components. Even though Volley examines data dependencies and CloneCloud, Cloudward Bound, and Manticore examine component or code dependencies, none of these approaches combine code and data dependencies to drive their partitioning and distribution decisions. In this paper, we demonstrated how combining code and data dependencies can provide a richer model that better supports cross-tier partitioning for web application in a hybrid architecture.

Database Partitioning: Database partitioning is generally divided into horizontal partitioning and vertical partitioning [7]. In horizontal partitioning, the rows of some tables are split across multiple hosts. A common motivation is for load-

balancing the database workload across multiple database manager instances [12, 23]. In vertical partitioning, some columns of the database are split into groups which are commonly accessed together, improving access locality [5]. Unlike traditional horizontal or vertical partitioning, our partitioning of data works at the granularity of entire tables. This is because our motivation is not only performance based but is motivated by policies on the management of data resources in the hybrid architecture. The granularity of logical tables aligns more naturally than columns with common business policies and access controls. That being said, we believe if motivated by the right use-case, our technical approach could likely be extended for column-level partitioning as well.

8 Limitations, Future Work, and Conclusion

While our approach simplifies manual reasoning for hybrid cloud partitioning, it requires some input from a developer. First, we require a representative workload for profiling. Second, a developer may need to provide input about the impact that atomic transactions have on partitioning. After partitioning, a developer may also want to consider changes to the implementation to handle some transactions in an alternative fashion, e.g. providing forward compensation [13]. Also as noted, our current implementation and experience is limited to Java-based web applications and SQL-based databases.

In future work we plan to support a more loosely coupled service-oriented architecture for partitioning applications. Our current implementation of data-tier partitioning relies on leveraging the distributed query engine from a production database. In some environments, relying on a homogeneous integration of data by the underlying platform may not be realistic. We are currently working to automatically generate REST interfaces to integrate data between the public cloud and private premise rather than relying on a SQL layer.

In this paper we have demonstrated that combining code and data dependency models can lead to cheaper and better performing hybrid deployment of Web applications. In particular, we showed that for our evaluated applications, combined code and data partitioning can achieve up to 56% performance improvement compared to a naïve partitioning of code and data between the cloud and the premise and a more than 40% performance improvement compared to when only code is partitioned (see Section 6.1). Similarly, for deployment costs, we showed that combining code and data can provide up to 54% expected cost savings compared to a fully premise deployment and almost 30% expected savings compared to a naïvely partitioned deployment of code and data or a deployment where only code is partitioned (cf. Section 6.2).

References

1. Apache DayTrader https://cwiki.apache.org/GMOxDOC20/daytrader.html.
2. lp_solve Linear Programming solver - http://lpsolve.sourceforge.net/.
3. Manticore Homepage - Online: http://nima.magic.ubc.ca/manticore.
4. RUBiS: Rice University Bidding System - Online: http://rubis.ow2.org/.
5. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *VLDB Jour.*, 18(2), 2009.

6. S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *Proc. of NSDI*, 2010.
7. S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proc. of (SIGMOD)*, 2004.
8. M. Armbrust, A. Fox, R. Griffith, and et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, UC Berkeley, 2009.
9. E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *Proc. of wsh. on Secure Web Services*, 2009.
10. S. Chong, J. Liu, A. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Building secure web applications with automatic partitioning. In *Proc. of SOSP*, 2009.
11. B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proc. of EuroSys*, 2011.
12. C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2), 2010.
13. H. Garcia-Molina and K. Salem. Sagas. In *Proc. of SIGMOD*, 1987.
14. M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. In *Proc. of SIGCOMM*, 2010.
15. G. Hunt and M. Scott. The Coign automatic distributed partitioning system. In *Proc. of Symp. on Operating Systems Design and Implementation (OSDI)*, 1999.
16. W. Iqbal, M. N. Dailey, and D. Carrera. SLA-driven dynamic resource management for multi-tier web applications in a cloud. In *CCGRID*, 2010.
17. N. Kaviani, E. Wohlstadter, and R. Lea. Manticore: A Framework for Partitioning of Software Services for Hybrid Cloud. In *Proc. of IEEE CloudCom*, 2012.
18. V. Khadilkar, M. Kantarcioglu, and B. Thuraisingham. Risk-Aware Data Processing in Hybrid Clouds. Technical report, University of Texas at Dallas, 2011.
19. S. Y. Ko, K. Jeon, and R. Morales. The HybrEx model for confidentiality and privacy in cloud computing. In *Proc. of HotCloud*, 2011.
20. F. Leymann, C. Fehling, R. Mietzner, A. Nowak, and S. Dustdar. Moving applications to the cloud: an approach based on application model enrichment. *Int. J. Cooperative Inf. Syst.*, 20(3):307–356, 2011.
21. Microsoft. The Economics of the Cloud. USA, November 2010.
22. R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *Proc. of NSDI*, 2009.
23. A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proc. of SIGMOD*, 2012.
24. J. Rellermeyer, O. Riva, and G. Alonso. AlfredO: an architecture for flexible interaction with electronic devices. In *Middleware*, 2008.
25. A. Schrijver. *Theory of Linear and Integer Programming.* Wiley & Sons, 1998.
26. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of SIGMOD*, 1979.
27. C. Stewart, M. Leventi, and K. Shen. Empirical examination of a collaborative web application. In *IISWC*, 2008.
28. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proc. of ECOOP*, pages 178–204. Springer-Verlag, 2002.
29. A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud conductor. In *Proc. of NSDI*, 2012.
30. F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *WWW*, 2006.
31. C. T. Yu and C. C. Chang. Distributed Query Processing. *Comp. Surv.*, 1984.