A Comparison of Flexible Schemas for Software as a Service

Stefan Aulbach[†] Dean Jacobs[$] Alfons Kemper[†] Michael Seibold[†]

[†]Technische Universität München, Germany
{stefan.aulbach, alfons.kemper, michael.seibold}@in.tum.de

[$]SAP AG, Walldorf, Germany
dean.jacobs@sap.com

ABSTRACT

A multi-tenant database system for Software as a Service (SaaS) should offer schemas that are flexible in that they can be *extended* for different versions of the application and dynamically *modified* while the system is on-line. This paper presents an experimental comparison of five techniques for implementing flexible schemas for SaaS. In three of these techniques, the database "owns" the schema in that its structure is explicitly defined in DDL. Included here is the commonly-used mapping where each tenant is given their own private tables, which we take as the baseline, and a mapping that employs Sparse Columns in Microsoft SQL Server. These techniques perform well, however they offer only limited support for schema evolution in the presence of existing data. Moreover they do not scale beyond a certain level. In the other two techniques, the application "owns" the schema in that it is mapped into generic structures in the database. Included here are XML in DB2 and Pivot Tables in HBase. These techniques give the application complete control over schema evolution, however they can produce a significant decrease in performance. We conclude that the ideal database for SaaS has not yet been developed and offer some suggestions as to how it should be designed.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; H.2.1 [**Information Systems**]: Database Management—*Logical Design*

General Terms

Design, Performance

Keywords

Multi-Tenancy, Software as a Service, Flexible Schemas, Extensibility, Evolution

1. INTRODUCTION

In the Software as a Service (SaaS) model, a service provider owns and operates an application that is accessed by many businesses over the Internet. A key benefit of this model is that, by careful engineering, it is possible to leverage economy of scale to reduce total cost of ownership relative to on-premises solutions. Common practice in this regard is to consolidate multiple businesses into the same database to reduce operational expenditures, since there are fewer processes to manage, as well as capital expenditures, since resource utilization is increased.

A multi-tenant database system for SaaS should offer schemas that are flexible in two respects. First, it should be possible to *extend* the base schema to support multiple specialized versions of the application, e.g., for particular vertical industries or geographic regions. An extension may be private to an individual tenant or shared by multiple tenants. Second, it should be possible to dynamically *evolve* the base schema and its extensions while the database is on-line. Evolution of a tenant-owned extension should be totally "self-service": the service provider should not be involved; otherwise operational costs will be too high.

This paper presents an experimental comparison of five techniques for implementing flexible schemas for SaaS. In three of these techniques, the database "owns" the schema in that its structure is explicitly defined in DDL:

Private Tables: Each tenant is given their own private instance of the base tables that are extended as required. In contrast, in all of the other mappings, tenants share tables. We take Private Tables as the experimental baseline.

Extension Tables: The extensions are vertically partitioned into separate tables that are joined to the base tables along a row ID column.

Sparse Columns: Every extension field of every tenant is added to its associated base table as a Sparse Column. Our experiments here use Microsoft SQL Server 2008 [1]. To implement Sparse Columns efficiently, SQL Server uses a variant of the Interpreted Storage Format [4, 7], where a value is stored in the row together with an identifier for its column.

Our experimental results show that these techniques perform well, however they offer only limited support for schema evolution. DDL commands over existing data, if they are supported at all, consume considerable resources and negatively impact performance. In the on-line setting, the ap-

plication must be given control over when and how bulk data transformations occur. An additional issue is that these techniques do not scale beyond a certain level.

In the other two techniques, the application "owns" the schema in that it is mapped into generic structures in the database:

XML: Each base table is augmented by a column that stores all extension fields for a tenant in a flat XML document. Since these documents necessarily vary by tenant, they are untyped. Our experiments here use pureXML in IBM DB2 [20].

Pivot Tables: Each value is stored along with an identifier for its column in a tall narrow table [2]. Our experiments here use HBase [11], which is an open source version of Google BigTable [6]. BigTable and HBase were originally designed to support the exploration of massive web data sets, but they are increasingly being used to support enterprise applications [14]. The Pivot Table mapping into HBase that we employ is consistent with best practices.

These two techniques give the application complete control over schema evolution, however our experimental results show that they can produce a significant decrease in performance from the baseline. For XML, the decrease is greatest for reads, which require parsing the untyped documents and reassembling typed rows. The decrease is proportional to the number of extension fields. For Pivot Tables, the decrease is more than an order of magnitude in some cases. Note that these results should not be taken as a negative statement about the quality of these systems, since they have not been optimized for our use case. Moreover, HBase is an early-stage open source project, not a mature commercial product. Our results are intended to give a general indication of the trade-offs in implementing flexible schemas.

Several major SaaS vendors have developed mapping techniques in which the application owns the schema. This approach has been elevated to a design principle whereby the application derives essential capabilities by managing the metadata itself [19, 23]. To achieve acceptable performance, these applications re-implement significant portions of the database, including indexing and query optimization, from the outside. We believe that databases should be enhanced to directly support the required capabilities.

Our experiments are based on a multi-tenant database testbed that simulates a simple but realistic Customer Relationship Management (CRM) service. The workload contains single- and multi-row create, read, and update operations as well as basic reporting tasks. The schema can be extended for individual tenants and it can evolve over time. Our previous work with a more limited version of this testbed (no extensions) showed that the performance of Private Tables degrades if there are too many tables [3]. This effect is due to the large amount of memory needed to hold the metadata as well as an inability to keep index pages in the buffer pool. In this paper, we create only a moderate number of tables, take Private Tables as the baseline, and use it to compare the other mappings.

This paper is organized as follows. Section 2 describes our multi-tenant database testbed and the CRM application that it simulates. Section 3 describes the schema mapping techniques. Section 4 presents the results of our experiments. Section 5 concludes that the ideal database for SaaS has not yet been developed and offers some suggestions as to how it should be designed.

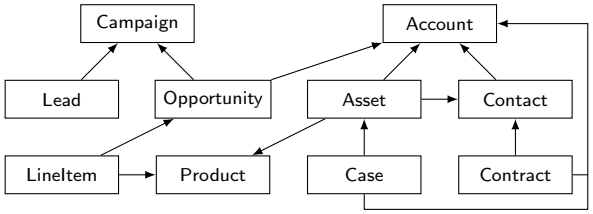


Figure 1: CRM Application Schema

2. MULTI-TENANT DATABASE TESTBED

The experiments in this paper are based on a multi-tenant database testbed we have developed that can be adapted for different database configurations. Each configuration requires a plug-in to the testbed that transforms abstract actions into operations that are specific to and optimized for the target database.

The testbed simulates a simple but realistic CRM service. Figure 1 shows the entities and relationships in the base schema. The base entities are extended with additional fields of various types for each tenant. Tenants have different sizes and tenants with more data have more extension fields, ranging from 0 to 100. The characteristics of the dataset are modeled on *salesforce.com*'s published statistics [13].

The testbed has nine request classes. The distribution of these requests is controlled using a mechanism similar to TPC's card decks.

Select 1: Select all attributes of a single entity as if it was being displayed in a detail page in the browser.

Select 50: Select all attributes of 50 entities as if they were being displayed in a list in the browser.

Select 1000: Select all attributes of the first 1000 entities as if they were being exported through a Web Services interface.

Reporting: Run one of five reporting queries that perform aggregation and/or parent-child-roll-ups.

Insert 1: Insert one new entity instance as if it was being manually entered into the browser.

Insert 50: Insert 50 new entity instances as if data were being synchronized through a Web Services interface.

Insert 1750: Insert 1750 new entity instances as if data were being imported through a Web Services interface.

Update 1: Update a single entity as if it was being modified in an edit page in the browser.

Update 100: Update 100 entity instances as if data were being synchronized through a Web Services interface.

The testbed mimics a typical application server's behavior by creating a configurable number of connections to the database backend. To avoid blockings, the connections are distributed among a set of worker hosts, each of them handling a few connections only. Distributing these connections among multiple hosts allows for modeling various sized, multi-threaded application servers.

3. SCHEMA MAPPING TECHNIQUES

Within a SaaS application, each tenant has a *logical* schema consisting of the base schema and a set of extensions. To implement multi-tenancy, the logical schemas from multiple tenants are mapped into one *physical* schema in the database. The mapping layer transforms queries against the logical schemas into queries against the physical schema so multi-tenancy is transparent to application programmers.

The physical schemas for the five mapping techniques studied in this paper are illustrated in Figure 2. The example data set used in this Figure is most clearly shown in the Private Tables mapping (Figure 2(a)). There are three tenants – 17, 35, and 42 – each of which has an Account table with Account ID (Aid) and Name fields. Tenant 17 has extended the Account table with two fields for the health care industry: Hospital and Beds. Tenant 42 has extended the Account table with one field for the automotive industry: Dealers. In the Extension Tables mapping (Figure 2(b)), the industry extensions are split off into separate tables that are joined to the base Account table using a new Row number column (Row). Tenants share the tables using a tenant ID column (Tenant). This section describes the other three mappings in more detail.

3.1 Sparse Columns in Microsoft SQL Server

Sparse Columns were originally developed to manage data such as parts catalogs where each item has only a few out of thousands of possible attributes. Storing such data in conventional tables with NULL values can decrease performance even with advanced optimizations for NULL handling. To implement Sparse Columns, SQL Server 2008 uses a variant of the Interpreted Storage Format [4, 7], where a value is stored in the row together with an identifier for its column.

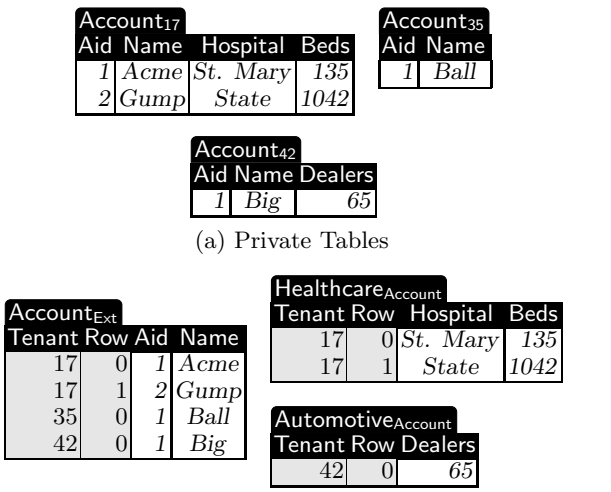
In our mapping for SaaS, the base tables are shared by all tenants and every extension field of every tenant is added to the corresponding base table as a Sparse Column, as illustrated in Figure 2(c). Sparse columns must be explicitly defined by a CREATE/ALTER TABLE statement in the DDL and, in this sense, are owned by the database. Nevertheless, the application must maintain its own description of the extensions, since the column names cannot be statically embedded in the code. For writes, the application must ensure that each tenant uses only those columns that they have declared, since the namespace is global to all tenants. For reads, the application must do an explicit projection on the columns of interest, rather than doing a SELECT $*$, to ensure that NULL values are treated correctly.

Sparse Columns requires only a small, fixed number of tables, which gives it a performance advantage over Private Tables; [3] shows that having many tables negatively impacts performance. On the other hand, there is some overhead for managing Sparse Columns. As an example, the SQL Server documentation recommends using a Sparse Column for an INT field only if at least 64% of the values are NULL [15]. Both of these factors are reflected in the performance results presented in Section 4.

3.2 XML in IBM DB2

IBM pureXML was designed to allow processing of semi-structured data alongside of structured relational data [20]. The mapping for SaaS that we use follows the recommendations in the pureXML documentation for supporting multi-tenancy [21]. The base tables are shared by all tenants and each base table is augmented by a column (Ext_XML) that stores all extension fields for a tenant in a flat XML document, as illustrated in Figure 2(d). Since these documents necessarily vary by tenant, they are untyped. This representation keeps the documents as small as possible, which is an important consideration for performance [16].

pureXML offers a hybrid query language that provides native access to both the structured and semi-structured representations. Our testbed manipulates data in the structured format, thus accessing extension data requires a correlated subquery to manage the XML. This subquery extracts the relevant extension fields using the XMLTABLE function which converts an XML document into a tabular format using XPath. The query with the XMLTABLE function has



Account17

Aid	Name	Hospital	Beds
1	Acme	St. Mary	135
2	Gump	State	1042

Account35

Aid	Name
1	Ball

Account42

Aid	Name	Dealers
1	Big	65

(a) Private Tables

AccountExt

Tenant	Row	Aid	Name
17	0	1	Acme
17	1	2	Gump
35	0	1	Ball
42	0	1	Big

HealthcareAccount

Tenant	Row	Hospital	Beds
17	0	St. Mary	135
17	1	State	1042

AutomotiveAccount

Tenant	Row	Dealers
42	0	65

(b) Extension Tables

Account

Tenant	Aid	Name	SPARSE
17	1	Acme	Hospital St. Mary Bed 135
17	2	Gump	Hospital State Bed 1042
35	1	Ball	
42	1	Big	Dealer 65

(c) Sparse Columns

Account

Tenant	Aid	Name	Ext_XML
17	1	Acme	<ext><hospital>St. Mary</hospital><beds>135</beds></ext>
17	2	Gump	<ext><hospital>State</hospital><beds>1042</beds></ext>
35	1	Ball	
42	1	Big	<ext><dealers>65</dealers></ext>

(d) XML

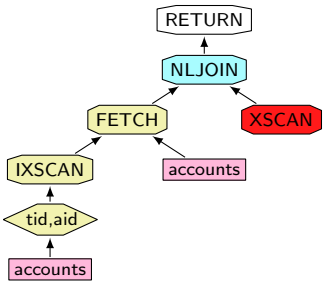
RowKey	Account	Contact
17Act1	[name:*Acme*, hospital:*St. Mary*, beds:*135*]	
17Act2	[name:*Gump*, hospital:*State*, beds:*1042*]	
17Ctc1		[···]
17Ctc2		[···]
35Act1	[name:*Ball*]	
35Ctc1		[···]
42Act1	[name:*Big*, dealers:*65*]	

(e) Pivot Tables

Figure 2: Schema Mapping Techniques

```
SELECT b.Tenant, b.Aid, b.Name,
       e.Dealers
FROM   Accounts b,
       XMLTABLE('i/ext' PASSING b.Ext_XML AS "i"
         COLUMNS
           Dealers INTEGER PATH 'dealers'
       ) AS e
WHERE Tenant = 42 AND Aid = 1;
```

(a) Physical SELECT Query



(b) Query Execution Plan

Figure 3: Correlated Subquery for XML in DB2

to be generated client- and query-specific to access clients' extension fields relevant in the particular query. Figure 3(a) shows an example query against the physical schema that selects three base fields and one extension field; Figure 3(b) shows the associated query plan. In our testbed, rows are always accessed through base fields, hence there is no need to use the special XML indexes offered by pureXML [20]. To insert a new tuple with extension data, the application has to generate the appropriate XML document; our performance results generally include the time to perform this operation. Updates to extension fields are implemented using XQuery 2.0 features to modify documents in place.

3.3 Pivot Tables in HBase

HBase [11], which is an open source version of Google BigTable [6], was originally designed to support the exploration of massive web data sets. These systems are increasingly being used to support enterprise applications in a SaaS setting [14].

In an HBase table, columns are grouped into *column families*. Column families must be explicitly defined in advance in the HBase "DDL", for this reason they are owned by the database. There should not be more than tens of column families in a table and they should rarely be changed while the system is in operation. Columns within a column family may be created on-the-fly, hence they are owned by the application. Different rows in a table may use the same column family in different ways. All values in a column are stored as Strings. There may be an unbounded number of columns within a column family.

Data in a column family is stored together on disk and in memory. Thus, a column family is essentially a Pivot Table; each value is stored along with an identifier for its column in a tall narrow table [2].

HBase was designed to scale out across a large farm of servers. Rows are range-partitioned across the servers by key. Applications define the key structure, therefore implicitly control the distribution of data. Rows with the same key

```
SELECT   p.Name, COUNT(c.Case_id) AS cases
FROM     Products p, Assets a, Cases c
WHERE    c.Asset = a.Asset_id
  AND    a.Product = p.Product_id
GROUP BY p.Name
ORDER BY cases DESC
```

Figure 4: Logical Reporting Query

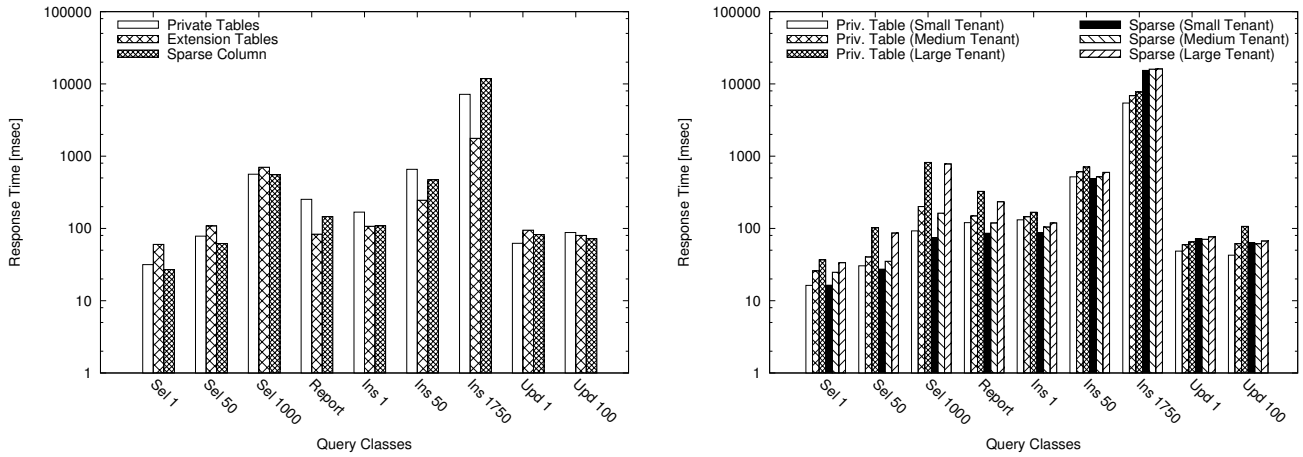
prefix will be adjacent but, in general, may end up on different servers. The rows on each server are physically broken up into their column families.

The mapping for SaaS that we use is illustrated in Figure 2(e). In keeping with best practices for HBase, this mapping ensures that data that is likely to be accessed within one query is clustered together. A single HBase table is used to store all tables for all tenants. The physical row key in HBase consists of the concatenation of the tenant ID, the name of the logical table, and the key of the row in the logical table. Each logical table is packed into its own column family, thus each row has values in only one column family. Within a column family, each column in the logical table is mapped into its own physical HBase column. Thus, since columns are dynamic, tenants may individually extend the base tables.

The reporting queries in our testbed require join, sort and group operations, which are not currently provided by HBase. We therefore implemented these operators outside the database in an *adaptation layer* that runs in the client. The adaptation layer utilizes operations in the HBase client API such as update single-row, get single-row and multi-row scan with row-filter. As an example, consider the reporting query shown in Figure 4, which produces a list of all Products with Cases by joining through Assets. To implement this query, our adaptation layer scans through all Cases for the given tenant and, for each one, retrieves the associated Asset and Product. It then groups and sorts the data for all Cases to produce the final result.

In our experiments, HBase was configured to run on a single node and the Hadoop distributed map-reduce framework was not employed. In our experience, hundreds of tenants for an application like CRM can be managed by a database on a single commodity processor. In this setting, spreading the data for a tenant across multiple nodes and doing distributed query processing would not be advantageous; the overhead for managing the distribution would nullify any benefits of parallelization. Of course, in addition to scaling up to handle many small tenants, the ideal SaaS database should also scale out to handle large tenants. But even in this case, map-reduce is problematic for queries such as the one in Figure 4, since it requires that data be clustered around Products. Other queries, such as pipeline reports on Opportunities, might require that the data be clustered in other ways.

We conclude this section with several comments about the usage of HBase in our experiments. First, HBase offers only row-at-a-time transactions and we did not add a layer to extend the scope to the levels provided by the commercial databases. Second, compression of column families was turned off. Third, neither major nor minor compactions occurred during any of the experiments. Fourth, replication of data in the Hadoop file system was turned off. Fifth, column families were not pinned in memory. Sixth, the system was configured so that old attribute values were not maintained.



(a) Overall (b) By Tenant Size

Figure 5: SQL Server Performance

4. EXPERIMENTAL RESULTS

This section presents the results of our experiments on schema extensibility and evolution. To study schema evolution, we issued a series of schema alteration statements during a run of the testbed and measured the drop in throughput. The experiments were run on Microsoft SQL Server 2008, IBM DB2 V.9.5 on Windows 2008, and HBase 0.19 on Linux 2.6.18 (CentOS 5.2). The database host was a VM on VMWare ESXi with 4 3.16 GHz vCPUs and 8 GB of RAM.

4.1 Microsoft SQL Server

Figure 5(a) shows the results of running our testbed on Microsoft SQL Server using three different mappings: Private Tables, Extension Tables, and Sparse Columns. The horizontal axis shows the different request classes, as described in Section 2, and the vertical axis shows the response time in milliseconds on a log scale.

In comparison to Private Tables, Extension Tables clearly exhibits the effects of vertical partitioning: wide reads (Sel 1, Sel 50, Sel 1000) are slower because an additional join is required, while narrow reads (Report) are faster because some unnecessary loading of data is avoided. Updates (Upd 1, Upd 100) perform similarly to wide reads because our tests modify both base and extension fields. Extension Tables is faster for inserts because tables are shared among tenants so there is a greater likelihood of finding a page in the buffer pool with free space.

Sparse Columns performs as well or better than Private Tables in most cases. The additional overhead for managing the Interpreted Storage Format appears to be offset by the fact that there are fewer tables. Sparse Columns performs worse for large inserts (Ins 1750), presumably because the implementation of the Interpreted Storage Format is tuned to favor reads over writes.

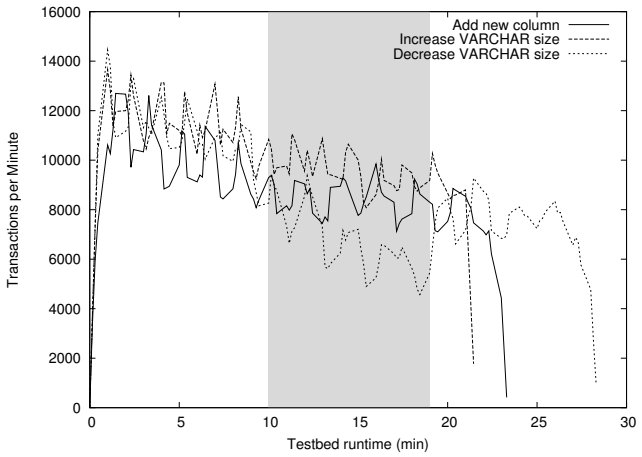
Figure 5(b) shows a break down of the Private Table and Sparse Column results by tenant size. Recall from Section 2 that larger tenants have more extension fields, ranging from 0 to 100. The results show that the performance of both mappings decreases to some degree as the number of extension fields goes up.

SQL Server permits up to 30,000 Sparse Columns per table. Our standard configuration of the testbed has 195 tenants, which requires about 12,000 columns per table. We also tried a configuration with 390 tenants and about 24,000 columns per table and there was little performance degradation. The number of extension fields per tenant in our testbed is drawn from actual usage, so SQL Server is unlikely to be able to scale much beyond 400 tenants. As a point of comparison, *salesforce.com* maintains about 17,000 tenants in one (very large) database [13].

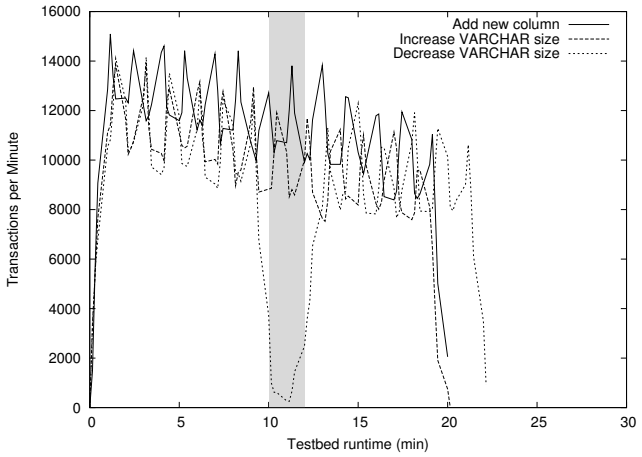
Figures 6(a) and 6(b) show the impact of schema evolution on throughput in SQL Server. In these graphs, the horizontal axis is time in minutes and the vertical axis is transactions per minute. The overall trend of the lines is downward because data is inserted but not deleted during a run. Part way through each run, ALTER TABLE statements on 5 base tables were submitted. The first two lines in each graph show schema-only DDL statements: add a new column and increase the size of a VARCHAR column. The third line in each graph shows a DDL statement that affects existing data: decrease the size of a VARCHAR column. To implement this statement, SQL Server scans through the table and ensures that all values fit in the reduced size. A more realistic alteration would perform more work than this, so the results indicate a lower bound on the impact of evolution. The gray bar on each graph indicates the period during which this third operation took place.

In the Private Tables case (Figure 6(a)), 975 ALTER TABLE statements were submitted, 5 for each of the 195 tenants. Individual schema-only alterations completed very rapidly, but nevertheless had an impact on throughput because there were so many of them. Adding a new column took about 1 minute to complete while increasing the size of a VARCHAR column took about 3 minutes. Decreasing the size of a VARCHAR column took about 9 minutes and produced a significant decrease in throughput. The overall loss of throughput in each case is indicated by the amount of time it took to complete the run.

In the Sparse Columns case (Figure 6(b)), the tables are shared and 5 ALTER TABLE statements were submitted. The schema-only changes completed almost immediately and had no impact on throughput. Decreasing the size of a VARCHAR column took about 2 minutes, during which through-

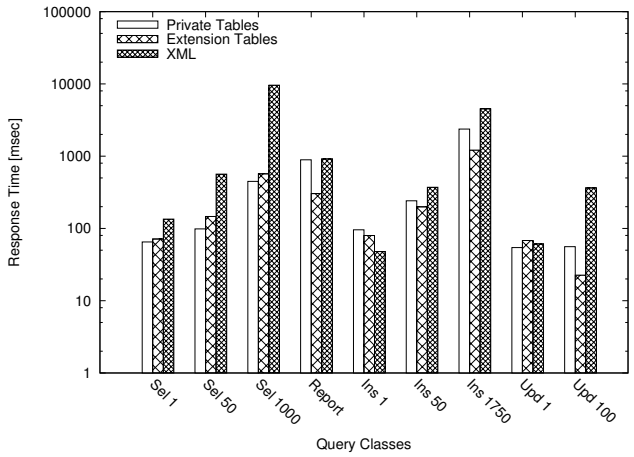


(a) Private Tables

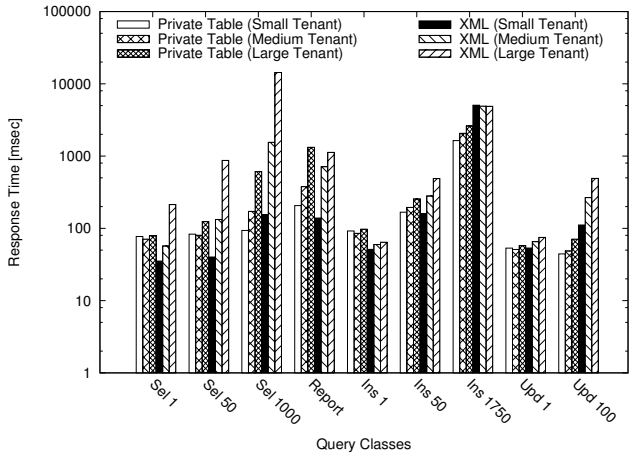


(b) Sparse Columns

Figure 6: SQL Server Throughput



(a) Overall



(b) By Tenant Size

Figure 7: DB2 Performance

put dropped almost to zero. The overall loss of throughput was greater for Private Tables, as indicated by the amount of time it took to complete the runs. However the behavior of Private Tables is probably preferable in the SaaS setting because the throughput drop is never as deep, thus the servers don't need to be overprovisioned as much. In any case, neither of these mappings is ideal in that the application should have more control over when such resource-intensive operations occur.

4.2 IBM DB2

Figure 7(a) shows the results of running our testbed on IBM DB2 using three different mappings: Private Tables, Extension Tables, and XML using pureXML. The axes are the same as in Figure 5.

In comparison to Private Tables, Extension Tables exhibits the same performance variations as in SQL Server. However XML produces a decrease in performance in most cases. The decrease is particularly severe for reads, which require executing a correlated subquery containing an XQuery statement embedded in a call to the XMLTABLE function, as described in Section 3.2. Figure 7(b) shows a break down of the Private Table and XML results by tenant size. Recall from Section 2 that larger tenants have more extension fields, ranging from 0 to 100. The results show that for reads, the performance decrease of XML is proportional to the number of extension fields. Note that in the Insert 1750 case, the results do not include the time to construct the XML document (for no particularly good reason) and there is no variation based on tenant size.

XML gives the application complete control over schema evolution. In this setting, the application is responsible for performing any bulk transformations associated with schema alterations that impact existing data. To study the efficiency of such transformations, we ran our schema evolution throughput experiment on DB2 using pureXML. To simulate the decrease-VARCHAR case, we submitted a query for each of the five base tables that SELECTs one field from all documents. These queries were run on the database server so no data transfer costs were incurred. The results were almost identical to the SQL Server Sparse Columns case shown in Figure 6(b). The five queries took about 2 minutes to complete, during which time throughput dropped to a very low level. Of course, the advantage of the XML mapping is that the application need not perform such transformations all at once.

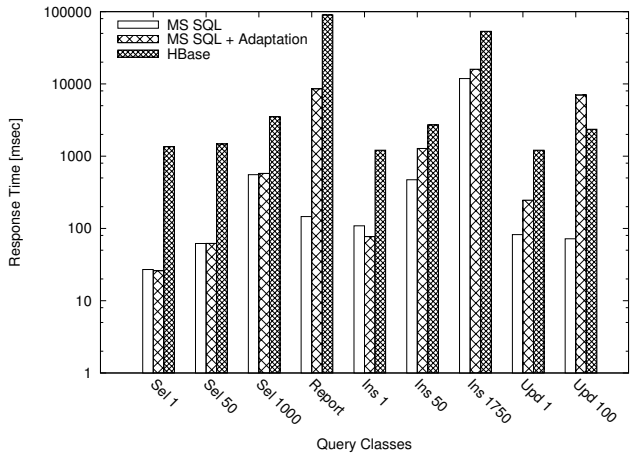


Figure 8: HBase Performance

4.3 HBase

Figure 8 shows the results of running our testbed on HBase along with two SQL Server configurations: one using the Sparse Columns mapping presented in Section 4.1 and one using the adaptation layer described in Section 3.3. Recall that the adaptation layer performs join, sort and group operations outside the database. In the latter case, to further approximate the HBase mapping, we made the base columns as well as the extension columns sparse. It turns out that this change was not significant: according to independent testbed runs we performed, making the base fields sparse has little impact on the performance of SQL Server.

In comparison to the Sparse Columns mapping in SQL Server, HBase exhibits a decrease in performance that ranges from one to two orders of magnitude depending on the operation. One reason for this decrease is the reduced expressive power of the HBase APIs, which results in the need for the adaptation layer. This effect is particularly severe for reports and updates, where SQL Server with adaptation also shows a significant decrease in performance. These results are consistent with [9], which shows that high-volume data-processing tasks such as reporting are more efficiently processed by shipping queries to the server rather than by shipping data to the client. The performance decrease for updates is primarily due to the fact that the adaptation layer submits changes one at a time rather than in bulk. HBase has a bulk update operation, however it appears that, in the version we used, changes are not actually submitted in bulk unless automatic flushing to disk is turned off. In any case, this problem with bulk updates should be easy to fix.

A second reason for the performance decrease is that rows must be assembled from and disassembled into Pivot Tables. Since the data for a row may be spread out across the disk, several reads or writes may be required. This effect is particularly severe for reads. A third reason is that HBase accesses disks (in the Hadoop File System) over the network. In contrast, shared-nothing architectures put disks on the local SCSI bus while shared-disk architectures use fast SANs.

5. CONCLUSIONS

The conclusion we draw from these experiments is that the ideal database system for SaaS has not yet been developed. The mappings in which the application owns the schema perform poorly unless significant portions of the database are re-implemented from the outside. And the mappings in which the database owns the schema provide only limited support for schema evolution in the presence of existing data. Moreover they cannot be scaled beyond a certain level.

We believe that the ideal SaaS database system should be based on the Private Tables mapping. The interleaving of tenants, which occurs in all of the other mappings, breaks down the natural partitioning of the data. It forces import and export of a tenant's data, which must occur for backup/restore and migration, to be carried out by querying the operational system. In contrast, with Private Tables, each tenant's data is clustered together on disk so it can be independently manipulated. The interleaving of tenant data also complicates access control mechanisms in that it forces them to occur at the row level rather than the table level.

In the ideal SaaS database system, the DDL should explicitly support schema extension. The base schema and its extensions should be registered as "templates". There should be multiple tenants and each tenant should be able to select various extensions to the base schema. The system should not just stamp out a new copy of the schema for each tenant, rather it should maintain the sharing structure. This structure should be used to apply schema alterations to all relevant tenants. In addition, this structure will reduce the amount of meta-data managed by the system.

In the ideal SaaS database system, the DDL should support on-line schema evolution. The hard part here is allowing evolution over existing data. As an example, Oracle implements this capability as follows: first a snapshot of the existing data is transformed into an interim table, then all transactions are blocked while any intervening updates are processed, and finally the original table is replaced by the interim table [17]. While this is a great step forward, the application must be given more control over when such resource-intensive operations occur. It should be possible to perform data transformations eagerly at the point the schema is changed, lazily at the point data is accessed, or every time data is accessed [18].

The ideal SaaS database system should distribute the data for many tenants across a farm of servers. The distribution should be *tenant-aware* rather than lumping all tenants into one large data set. It should be possible to spread the data for a large tenant across multiple servers, but that will not be the common case and in practice should not require more than a handful of servers per tenant. To support data distribution for large tenants, the query language should be less powerful than in conventional databases, however it should not be so weak that the common case suffers. In particular, the database should support multiple communication patterns for joins, rather than requiring the use of map-reduce. For example, joins in OLAP queries on a star schema tend to distribute well because the dimension tables are small.

The ideal SaaS database system should have a shared-nothing architecture where data is stored on fast local disks. Data should be explicitly replicated by the database, rather than a distributed file system, and used to provide high availability. To facilitate failure handling, the transactional guarantees should be weaker than in conventional databases [8]. In addition to supporting high availability, replicated data should be used to improve scalability (handle more requests for a given set of data), improve performance (maintain the

data in several different formats), and support on-line upgrades (roll upgrades across the replicas).

This paper has focused on conventional row-oriented databases for OLTP. Enterprise applications also require column-oriented databases for OLAP [5, 12, 22]. The basic prescription for SaaS databases offered here – Private Tables with support for on-line schema extension and evolution – applies to column stores as well as row stores.

The vertical storage structures of HBase and BigTable, which we use to implement Pivot Tables, are similar to column stores in that they are designed for narrow operations over many rows. Such vertical structures may be made more competitive for wide operations by keeping the data in memory, since the cost of reassembling rows is dominated by the time to perform many reads from disk. Advancements in data storage technologies are gradually making main-memory databases more attractive [10]. Our basic prescription for SaaS applies equally well to main-memory databases.

6. REFERENCES

[1] S. Acharya, P. Carlin, C. A. Galindo-Legaria, K. Kozielczyk, P. Terlecki, and P. Zabback. Relational Support for Flexible Schema Scenarios. *PVLDB*, 1(2):1289–1300, 2008.

[2] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In *VLDB*, pages 149–158, 2001.

[3] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *SIGMOD*, pages 1195–1206, 2008.

[4] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs To Support Sparse Datasets Using an Interpreted Attribute Storage Format. In *ICDE*, page 58, 2006.

[5] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Phd, University of Amsterdam, The Netherlands, 2002. `http://old-www.cwi.nl/htbin/ins1/publications?request=abstract&#38;key=Bo:DISS:02`.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[7] E. Chu, J. Beckmann, and J. Naughton. The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets. In *SIGMOD*, pages 821–832, 2007.

[8] S. Finkelstein, D. Jacobs, and R. Brendle. Principles for Inconsistency. In *CIDR*, 2009.

[9] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *SIGMOD*, pages 149–160, 1996.

[10] J. Gray. Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King. `http://research.microsoft.com/~Gray/talks/Flash_Is_Good.ppt`, 2006.

[11] HBase. `http://hadoop.apache.org/hbase/`, 2009.

[12] T. Legler, W. Lehner, and A. Ross. Data Mining with the SAP Netweaver BI Accelerator. In *VLDB*, pages 1059–1068, 2006.

[13] T. McKinnon. Plug Your Code in Here – An Internet Application Platform. `www.hpts.ws/papers/2007/hpts_conference_oct_2007.ppt`, 2007.

[14] MegaStore. `http://perspectives.mvdirona.com/2008/07/10/GoogleMegastore.aspx`, 2009.

[15] Microsoft SQL Server 2008 Books Online – Using Sparse Columns. `http://msdn.microsoft.com/en-us/library/cc280604.aspx`.

[16] M. Nicola. 15 Best Practices for pureXML Performance in DB2. `http://www.ibm.com/developerworks/data/library/techarticle/dm-0610nicola/`, 2008.

[17] Oracle Database 10g Release 2 Online Data Reorganization & Redefinition. `http://www.oracle.com/technology/deploy/availability/pdf/ha_10gR2_online_reorg_twp.pdf`.

[18] J. F. Roddick. Reduce, Reuse, Recycle : Practical Approaches to Schema Integration, Evolution and Versioning. *Lecture Notes in Computer Science*, 37:209–216, 2006.

[19] Salesforce.com: The Force.com Multitenant Architecture. `http://www.apexdevnet.com/media/ForcedotcomBookLibrary/Force.com_Multitenancy_WP_101508.pdf`.

[20] C. M. Saracca, D. Chamberlin, and R. Ahuja. *DB2 9: pure XML – Overview and Fast Start*. IBM, `http://ibm.com/redbooks`, 2006.

[21] M. Taylor and C. J. Guo. Data Integration and Composite Business Services, Part 3: Build a Multi-Tenant Data Tier with Access Control and Security. `http://www.ibm.com/developerworks/db2/library/techarticle/dm-0712taylor/`, 2007.

[22] Vertica. `http://www.vertica.com/`, 2009.

[23] Workday: Forget ERP, Start Over. `http://blogs.zdnet.com/SAAS/?p=368`.