UNIVERSITY OF TWENTE.

Multi-Tenant Customizable Databases

Master's Thesis

Wim van der Zijden

Supervisors:
dr. ir. D. Hiemstra (University of Twente)
dr. ir. M. van Keulen (University of Twente)
bc. S.M. van Dijk (ActFact Projects B.V.)
ing. E. Tomassen (ActFact Projects B.V.)

Enschede, February 2017

Voorwoord / Preface

Met het opleveren van deze scriptie is een einde gekomen aan een traject van viereneenhalf jaar waarin ik naast mijn werk als software engineer bij ActFact anderhalve bachelor, een universitaire master en een excellentie programma voltooid heb.

Ik ben enorm veel dankbaarheid verschuldigd aan mijn werkgever, die mij in staat heeft gesteld deze opleidingen te volgen en altijd buitengewoon flexibel is omgesprongen met de vaak sterk wisselende collegetijden en examenweken.

Deze masterscriptie is, net als mijn bachelorscriptie, uitgevoerd bij deze zelfde werkgever. Het vormt de aanzet voor een ambitieus en dapper project, waarvan de uitkomst allesbehalve zeker is. Hopelijk zal deze thesis een succesvolle kickstart kunnen betekenen.

Ik wil Eldir Tomassen bedanken, mijn technisch directeur en de geestelijk vader van deze opdracht. Daarnaast ben ik ook Sjoerd van Dijk dank verschuldigd voor zijn kritische blik en het behartigen van de bedrijfsbelangen. Ook wil ik graag Djoerd Hiemstra bedanken voor het waarborgen van het academische niveau van dit praktisch ingestoken onderzoek. En ten slotte heeft Maurice van Keulen vooral in de eindfase van dit project een zeer positieve invloed gehad op de kwaliteit van het eindresultaat door zijn onbevangen, kritische blik.

In plaats van een traditionele master thesis, is deze thesis een bundeling van twee wetenschappelijke papers geworden, waarvan ik de eerste in ingekorte vorm zal indienen bij de database conferentie ADBIS 2017.

This thesis marks the end of a four-and-a-half year period in which I completed one-and-a-half bachelor's degree, a master's degree and an honours programme while working as a sofware engineer at ActFact.

I owe a debt of gratitude to my employer for enabling me to attend these studies and for their remarkable flexibility with regards to the often highly variable lecture times and exam weeks.

This master's thesis was, as my bachelor's thesis, carried out at this same employer. It is the preliminary for an ambitious and brave project, of which the conclusion is all but certain. Hopefully this thesis can act as a successful kickstart.

I want to thank Eldir Tomassen, my technical director and spiritual father of this assignment. I also owe a debt of gratitude to Sjoerd van Dijk, for his critical view and for governing the company's interests. Furthermore, I would like to thank Djoerd Hiemstra for maintaining the academic level in this practice oriented research. Finally, especially in the end phase of this project, Maurice van Keulen had a very positive influence on the quality of this work with his open-minded, critical view.

Instead of a traditional master's thesis, this thesis has become a bundle of two research papers, of which I will submit the first one in an abridged version at the database conference ADBIS 2017.

Introduction

A good practice in business is to focus on key activities. For some companies this may be branding, while other businesses may focus on areas such as consultancy, production or distribution. Focusing on key activities means to outsource as much other activities as possible. These other activities merely distract from the main goals of the company and the company will not be able to excel in them.

Many companies are in need of reliable software to persistently process live data transactions and enable reporting on this data. To fulfil this need, they often have large IT departments in-house. Those departments are costly and distract from the company's main goals. The emergence of cloud computing should make this no longer necessary. All they need is an internet connection and a service contract with an external provider.

However, most businesses are in need of highly customizable software, because each company has slightly different business processes, even those in the same industry. So even if they outsource their IT need, they will still have to pay expensive developers and business analysts to customize some existing application.

These issues are addressed by Multi-Tenant Customizable (MTC) applications. We define such an application as follows:

> *A single software solution that can be used by multiple organizations at the same time and which is highly customizable for each organization and user within that organization, by domain experts without a technical background.*

A key challenge in designing such a system is to develop a proper persistent data storage, because mainstream databases are optimized for single tenant usage.

To this end this Master's thesis consists of two papers: the first paper proposes an MTC-DB Benchmark, *MTCB*. This Benchmark allows for objective comparison and evaluation of MTC-DB implementations, as well as providing a framework for the definition of MTC-DB. The second paper describes a number of MTC-DB implementations and uses the benchmark to evaluate those implementations.

MTCB: A Multi-Tenant Customizable database Benchmark

Wim van der Zijden

University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands
ActFact Projects, Josink Esweg 8, 7545 PN Enschede,The Netherlands
w.vanderzijden@actfact.com

Abstract. We argue that there is a need for Multi-Tenant Customizable OLTP systems. Such systems need a Multi-Tenant Customizable Database (MTC-DB) as a backing. To stimulate the development of such databases, we propose the benchmark MTCB. Benchmarks for OLTP exist and multi-tenant benchmarks exist, but no MTC-DB benchmark exists that accounts for customizability. We formulate seven requirements for the benchmark: realistic, unambiguous, comparable, correct, scalable, simple and independent. It focuses on performance aspects and produces nine metrics: aulbach compliance, size on disk, tenants created, types created, attributes created, transaction data type instances created per minute, transaction data type instances loaded by ID per minute, conjunctive searches per minute and disjunctive searches per minute. We present a specification and an example implementation in Java 8, which can be accessed on this public repository: `https://bitbucket.org/actfact/mtcdb-benchmark`. In the same repository an implementation can be found for a naive implementation of an MTC-DB where each tenant has its own schema. We believe that this benchmark is a valuable contribution to the community of MTC-DB developers, because it provides objective comparability as well as a precise definition of the concept of MTC-DB.

Keywords: Multi-Tenant Customizable, Multi-level customizability, OLTP, PaaS, Database, Benchmark

1 Introduction

1.1 Context

A good practice in business is to focus on key activities. For some companies this is mostly branding their product [1]. Other businesses may focus on areas such as consultancy, production or distribution. Focusing on key activities means to outsource as much other activities as possible. These other activities merely distract from the main goals of the company and the company will not be able to excel in them.

Many companies are in need of OLTP[1] software. To fulfil this need, they often have large IT departments in-house. Those departments are costly and distract from the company's main goals. The emergence of cloud computing should make this no longer necessary. All they need is an internet connection and a service contract with an external provider.

However, most businesses are in need of highly customizable software, because each company has slightly different business processes, even those in the same industry. So even if they outsource their IT need, they will still have to pay expensive developers and business analysts to customize some existing OLTP application. A large problem is the communication gap [2]: most developers do not understand the business domain, and most domain experts do not understand the technical implications of their requirements.

These issues are addressed by Multi-Tenant Customizable (MTC) applications. We define such an application as follows:

> *A single software solution that can be used by multiple organizations at the same time and which is highly customizable for each organization and user within that organization, by domain experts without a technical background.*

A key challenge in designing such a system is to develop a proper persistent data storage, because mainstream databases are optimized for single tenant usage. To stimulate the development of such systems we will propose a benchmark to compare implementations of such a data storage, the Multi-Tenant Customizable database Benchmark: MTCB.

1.2 Problem Statement

To illustrate the main issues with traditional on-premises OLTP applications we will consider the open source ERP[2] and CRM[3] system Compiere[4] as a quintessential example. Similar systems such as SAP and Microsoft Dynamics Navision cope with similar problems.

Compiere's strong suit is its Model Driven Architecture. This means that the model of the application is defined in the Application Dictionary. The most important elements in the Application Dictionary are Table, Column, Window, Tab and Field. From the definitions in these elements, the user interface for the application is built. The main advantage of this approach is that it is relatively

[1] Online Transaction Processing. Applications whose main concern it is to persistently and reliably process live data transactions and to facilitate reporting on this data. An OLTP application is usually backed by a relational database.

[2] Enterprise Resource Planning. A type of business software that consolidates many business processes such as product planning, sales, inventory management and accounting.

[3] Customer Relationship Management. A type of business software that allows companies to collect, analyse and act on customer data in order to improve sales.

[4] `http://www.compiere.com/`

easy to add custom entities and fields to the application, according to business needs.

However, there are also some problems with this architecture. We have identified the following problems:

1. **Tedious installation procedure.** An expert is needed to properly configure and install Compiere. An application server and a database server must be set up. Normally this must be done twice, in order to have both a test and a production environment.
2. **Tedious maintenance of customizations.** An expert is needed to maintain customizations. In theory it is possible to customize the application on the fly, but this is very risky. It is adviced to develop new functionality in a development environment, then use a migration tool to deploy these customizations to the test environment, and after proper testing, release them to the production environment. This practice allows for proper versioning of customizations.
3. **Tedious development of customizations.** An expert is needed to develop customizations. In theory a non-technical expert user should be capable of making small customizations, but this is very limited. For example, to create custom processes, before and after save logic and callout code, it is necessary to write Java code, which needs to be integrated into the application using a migration tool.
4. **Entanglement of the data model and the user interface.** The business logic is implemented both on the user interface and in the data model. This makes it hard to develop alternative user interfaces on the system, because many business rules would have to be re-implemented. For example, read only logic on fields uses context variables which only exist in the context of a certain user interface. Especially callout logic is very much intertwined with the user interface. Compiere currently supports two user interfaces, a desktop and a web based one, for each of which the callout logic is entirely duplicated.
5. **No multi-tenant customizability.** To mitigate the overhead sketched above, a solution could be to use Compiere's multi-tenant setup. This way costs can be shared across multiple organizations. However, because Compiere is not multi-level customizable, this means that organizations would have to share customizations. So this will normally only be feasible if the organizations are active in the same industry, have no conflicting interests and are capable of aligning their business processes with each other.

1.3 Application requirements

Based on the concept of MTC, defined in section 1.1, we propose a radical solution to these problems: an OLTP *PaaS* that is *Single instance, Multi-Tenant, Modular, Multi-Level Customizable* and *Multi-Interface.* Each of these italicized terms is explained in the application requirements below.

1. **PaaS.** Platform as a Service. A cloud computing service that allows customers to develop and host applications without having to worry about many low-level concerns such as building the application and managing the infrastructure needed to host it. By defining the solution as a PaaS, we indicate that there is very little core functionality, and the application should be seen as a basic platform for building OLTP applications.

2. **Single-instance.** The application should functionally be one application in maintenance. This means that updating the application is a single action. However, in reality, it may be that the application is hosted over multiple servers and multiple databases, as long as this remains hidden for the developers on the platform.

3. **Multi-tenant.** A single instance of the application should be able to host a large number of tenants. A tenant is an organization that is using the application. As a guideline we say that the application should be able to host about a 1,000 tenants: 10 large tenants (50 concurrent users), 100 medium sized tenants (5 concurrent users) and 1,000 small and single-use tenants (0 to 1 concurrent user). Instead of a tedious installation procedure that requires hiring a professional to perform it, this system should offer a "single-click" installation procedure: creating a tenant should be as easy as creating an email account. This will mitigate problem 1: the tedious installation procedure.

4. **Modular.** Everything that is developed on the platform is developed as a module. When creating a new tenant, existing modules can be reused. When developing specialized modules for specific business needs, these modules may be shared with other tenants with similar business needs in order to cut down on development costs. This mitigates problems 2 and 3: tedious maintenance and development of customizations.

5. **Multi-level customizable.** The application should be customizable on several levels. Most importantly, the application may be customized on the organization level. So it needs to be possible for organizations to create customizations that are only visible on the organization level. Second, the application may be customized on user role level. So it needs to be possible to create customizations that are only visible when logged in as certain user roles. And third, the application may be customized on the level of the individual user. Customizability should be understood in a very broad sense. It may refer to the customization of entities, fields, windows, processes, reports, etc. This mitigates problem 5: no multi-tenant customizability.

6. **Multi-interface.** The model of the application should not be entangled with the interfaces it may offer to interact with it. For example, it should be relatively easy to implement alternative client side user interfaces using different JavaScript frameworks. However, the word interface as used here is not limited to user interfaces, but applies to Application Programming Interfaces (APIs) as well. This mitigates problem 4: entanglement of the data model and the user interface.

1.4 Technical Research Problem

The first challenge of designing an application as outlined above, is designing a suitable data storage structure. Currently no standalone databases exist that satisfy the requirement for such a data storage. To stimulate the development of such databases, we will design an MTC-DB benchmark. To this end we formulate the following technical research problem, using the template as proposed by Wieringa [3]:

> ***Improve*** *the evaluation of Multi-Tenant Customizable Database (MTC-DB) implementations,* ***by developing*** *an MTC-DB Benchmark specification* ***such that*** *it is realistic, unambiguous, comparable, correct, scalable, simple and independent* ***in order to*** *enable MTC-DB developers to assess the quality of their implementations objectively and use the result to advertise their solution.*

2 Existing Benchmarks

In this section we discuss existing benchmarks that are close to the benchmark that we are developing. TPC-C and TPC-E are OLTP benchmarks for single-tenant OLTP workloads. TPC-W is a discontinued benchmark for e-Commerce implementations that was extended by Krebs et al. [4] to account for multi-tenancy.

TPC stands for Transaction Processing Performance Council. This organization was founded in 1988 by eight companies that agreed that there was a need for a single standards body to supervise and govern benchmarks for transaction processing applications [5]. Since then, they published many benchmarks, of which three are specifically targeted at OLTP: TPC-A, TPC-C and TPC-E. TPC-A was their first benchmark and was made obsolete in 1995. TPC-C was approved in 1992, but the latest revision, 5.11, was published in 2010. TPC-E was approved in 2007 and was last revised in 2015. Even though TPC-E targets the same kind of environments, TPC-E did not formally make TPC-C obsolete.

2.1 TPC-C

The current revision of TPC-C is described in [6]. Some of the important characteristics of the environments that it aims to simulate are:

1. Various transactions are executed concurrently.
2. Transactions must adhere to the ACID properties.
3. The system consists of many "tables"[5] that differ greatly in size, attributes and relationships.

[5]The specification stresses that terms as table, row and column are meant to illustrate similar data structures as found in SQL-based databases, but may be implemented differently.

To this end, the specification describes a fictional data model of a wholesale supplier that has tables like warehouse, district and customer. In total the model consists of 9 tables. It describes in detail which attributes each table should have and what initial master data should be present in the system.

On this datamodel, a number of transaction profiles are described: the new-order transaction, the payment transaction, the order status transaction, the delivery transaction and the stock-level transaction. A complete business cycle uses a combination of these transactions to simulate the flow of a business process.

TPC-C produces four primary metrics:

1. **tpmC**: the business throughput per minute, measured as the number of processed orders.
2. **price per tpmC**: the total cost of running the system for 3 years, divided by the tpmC.
3. **Availability date**: the earliest date at which all components of the system will be available.
4. **watts per KtpmC**: the cost in energy per 1,000 tpmC (optional).

2.2 TPC-E

TPC-E is described in [7]. Both in goal and specification it is very similar to TPC-C. The main difference is that it is much more complex than TPC-C and claims to be more realistic and have a broader coverage. Its datamodel is that of a fictional brokerage firm. Chen et al. [8] summarize the differences as shown in Table 1. As can be seen, TPC-E specifies more tables, more attributes, check constraints and also tests for referential integrity.

	TPC-C	**TPC-E**
Tables	9	33
Columns	92	188
Columns per Table	10.2	5.7
Read-Write Transactions	92%	23.1%
Read-Only Transactions	8%	76.9%
Data generation	Random	Pseudo-Real
Check Constraints	0	22
Referential Integrity	No	Yes

Table 1. Comparison of TCP-C and TPC-E, based on Chen et al. [8]

2.3 Multi-Tenant TPC-W

TPC-W is another benchmark by the TPC. This benchmark is for e-Commerce web applications. However, it was made obsolete in 2005 and its specification is no longer available on tpc.org. The only original description that is still available is a whitepaper by Smith [9]. We were not able to find the reason that the TPC-W was made obsolete. A reason could be that web applications have evolved so much that it was simply not representative anymore. For example, the TPC-W is based on HTML 1.0 and does not account for Javascript and CSS.

TPC-W emulates a fictional online bookstore. It features 14 web pages that allow users to browse, search, order and pay for products. The test is run by using emulated browsers that simulate real users by employing random wait times between 7 and 70 seconds.

TPC-W produces two primary metrics:

1. **WIPS**: the number of web interactions per second that can be sustained by the system.
2. **Cost per WIPS**: the cost of the system divided by the WIPS rate.

Kreb et al. [4] enhanced TPC-W to make it into a Multi-Tenant benchmark. To this end, they made two important changes:

1. Adding a column "tenantId" to every table.
2. Adding a central administration mechanism for assigning primary keys.

Unfortunately, their extension does not provide clear comparable performance metrics. The results are graphs that show how increasing tenants impacts the original TPC-W metrics. Another problem with this benchmark is that it does not account for customizability.

3 MTC-DB Benchmark (MTCB)

In this section we describe the MTC-DB Benchmark (MTCB) that we developed. First we discuss the requirements we formulated, then the specification that we designed and finally the concrete steps a developer should take to start using the benchmark to implement and benchmark their own MTC-DB implementation.

3.1 Requirements

The design of MTCB has the following requirements:

- **R1 Realistic.** It should be a realistic reflection of an MTC environment in terms of performance aspects.
- **R2 Unambiguous.** It should be as unambiguous as possible. An MTC-DB developer should be able to implement it, purely based on the specification.
- **R3 Comparable.** It needs to provide objective and easily comparable performance metrics.

R4 **Correct.** It needs to test for correctness. This mainly concerns the ACID properties of database transactions: they should be atomic, consistent, isolated and durable.

R5 **Scalable.** It needs to be easy to benchmark very small scenarios, very large scenarios and a number of in between scenarios.

R6 **Simple.** It should be as simple as possible. Every added complexity should aid one of the other requirements in some way.

R7 **Independent.** The specification should stand on its own. It needs to be independent from any MTC-DB implementations. For example, there should be no dependency on the paradigm of SQL and/or RDBMS.

3.2 Conceptual Model

MTCB not only defines the metrics to measure the performance of MTC-DB implementations, it also defines *what* an MTC-DB implementation is. Our definition of MTC-DB is pure: there must be as little core functionality as possible. For example, the Force.com platform [10] is an MTC application that has a large amount of core functionality. For this core functionality they use traditional tables and only for the modifications by third party platform developers generic extension tables are used. In this model, the platform developers are second class to the native Force.com developers.

We defined the absolute minimum of core functionality to be four complex types[6] and four primitive types[7]. The complex types are tenant, user, type and attribute. The primitive types are string, number, timestamp and boolean. Some of these concepts are explained in detail below.

Tenant and User Bezemer et al. [11] give the following definition of *tenant*: "A tenant is the organizational entity which rents a multi-tenant SaaS solution. Typically, a tenant groups a number of users, which are the stakeholders in the organization."

This is a good basic definition, but in our model, we use an enhanced version of this definition. A tenant as described in that definition is what we call a data tenant. This kind of tenant mostly contains transactional data and master data, and little to no metadata. Its data is also isolated: no other tenants can access it.

A second type of tenant is a module tenant. This kind of tenant contains little transactional data and no master data, and instead only metadata. Furthermore, its data is not isolated: it is accessible by all tenants that have declared a dependency on it. It can also be dependent on other module tenants itself. This extension of the definition is necessary, because metadata needs to be shareable. If metadata could not be shared, then each data tenant would have to build its own application from scratch.

[6]Complex types are types that have attributes

[7]Primitive types are types that have no attributes

Type and Attribute Types are the metadata building blocks of the system. A type has a name and a display name, is defined within a tenant and has some attributes. Types can be also be enhanced with additional attributes by tenants other than the tenant that owns them.

Attributes refer to two types: their master type and their data type. The master type is the type that they are an attribute of, and the data type is the type of the data that they store. This can be one of the primitive types, but it can also be another complex type. An attribute must also indicate if it is searchable. This determines whether the attribute can be used in the predicate of a search query, to allow the MTC-DB implementation to optimize for this.

Search Design One of the things that should be benchmarked is how fast search queries run in the system. We distinguish two types of queries: load by ID and attribute search.

Load by ID Load by ID queries simulate the major workload of OLTP systems. In typical OLTP systems that use an RDBMS for storage, a window in the user interface of displays one row of a particular database table. To load this data, the application must retrieve this row. If this row contains foreign key references to other tables, then these must also be resolved.

In regular SQL, this can be done with a query of the following form:

```
SELECT t.ID, t.Name, t1.Name, t2.Name
FROM t,
JOIN t1 ON t1.ID = t.t1_ID
JOIN t2 ON t2.ID = t.t2_ID
WHERE t.ID = 1000;
```

To benchmark these type of queries, we specify Transaction Data Types (TDT) and Master Data Types (MDT). Both are synthetic data types that are also used for benchmarking customizability and creation of new type instances. An MDT is a type that contains only simple attributes. A TDT also contains complex types: references to MDTs.

Attribute search To benchmark queries with search predicates, we need to design search data and search queries. The search queries need to be representative for worst-case scenarios in the system and the search data needs to be generated automatically and needs to be scalable.

To this end we defined a designated search type and a designated search tenant, which are created in the Setup script (see section 3.3). This script also creates a number of instances of the search type in the search tenant, dependent on the Profile (see section 3.3).

We defined two distinct attribute searches: a conjunctive search and a disjunctive search. In regular single-tenant SQL, these searches have the following form.

Conjunctive search:

```
SELECT * FROM t
WHERE a1=? AND a2=? AND a3=? AND a4=? AND a5=?
```

Disjunctive search:

```
SELECT * FROM t
WHERE a6=? OR a7=? OR a8=? OR a9=? OR a10=?
```

We chose these two queries because these are two extremes and if an MTC-DB can implement these queries, they can implement all single type queries, because any propositional formula can be translated into the conjunctive normal form (CNF).

The search type has 10 attributes, 5 that are used by the conjunctive search, and 5 that are used by the disjunctive search. When instantiating these types, the attributes for the conjunctive search are populated with random integers in the range $[1, \sqrt[5]{n}]$, and the attributes for the disjunctive search are populated with random integers in the range $[1, 5n]$ where n is the total number of search type instances. Similarly, when creating the search queries, the search terms are randomly picked from the same range.

These ranges were picked specifically to make sure that no matter how large n gets, the searches will have approximately the same probability of returning no results, namely $\frac{1}{e}$, about 37%. This is true, because the probability for returning no results for the respective queries can be expressed with the following formulas, which both approximate $\frac{1}{e}$ for $\lim n \to \infty$.

Conjunctive Search:

$$P(n) = (1 - (\frac{1}{\sqrt[5]{n}})^5)^n$$

Disjunctive Search:

$$P(n) = (\frac{5n-1}{5n})^{5n}$$

3.3 Specification

MTCB consists of two main parts: the model and the scripts. The model consists of the contract of six interfaces. Every MTC-DB implementation must provide implementations for these interfaces. The scripts uses these interfaces to run the benchmark and report the performance metrics.

The model The model consists of six interfaces: `MTCDB`, `PO`, `Type`, `Attribute`, `Tenant` and `User`. The most important elements of the contract of those interfaces is shown in tables 2 through 5.

`MTCDB` is the main entry point. An instance of this class must be fed to the scripts. The most important operations that are defined on `MTCDB` are `createTenant()`, `createType()`, `createAttribute()` and `createPO()`.

PO stands for Persistent Object. This is the base contract for all entities that must be stored persistently. Its most important operations are persist() and several GetValueAs...() operations to retrieve a value for an attribute by attribute name.

MTCDB Interface	
addModule()	Add a metadata module to the tenant in the context.
createAttribute()	Add a new attribute to an existing type.
createPO()	Create a new instance of an existing type.
createTenant()	Create a new empty tenant.
createType()	Create a new type without attributes.
createUser()	Create a new user.
getByConjunction()	Retrieve PO instances by conjunction (AND) of attributes and values.
getByDisjunction()	Retrieve PO instances by disjunction (OR) of attributes and values.
getPOByID()	Retrieve a PO instance by ID.
getSizeOnDisk()	Retrieves the total size on disk in MB of the MTC-DB.
getTenant()	Get tenant by name.
getTenants()	Get all tenants that exist in the system.
getType()	Retrieve type by tenant and name.
getTypes()	Get all types for a tenant.
initializeSystem()	Set up the most elemental metadata necessary for the system.

Table 2. The contract for the interface MTCDB

Type, Attribute, Tenant and User are interfaces that extend PO. So they contain all the operations that PO contains, including some extra operations. For example, Type has the operation getAttributes() to get all attributes for that type, and Attribute has the operations getMasterType() to get the type it is an attribute of, and getDataType() to get the type of the value that it can store.

The scripts There are three scripts: the aulbach script, the setup script and the main script. These scripts are explained below. As input, each of these scripts needs an implementation instance of MTCDB. The latter two also need a profile that contains a set of parameters. How MTCDB is implemented and instantiated is up to the specific MTC-DB implementation.

PO Interface	
getID()	128-bit universally unique identifier for this instance as defined in RFC4122: `https://tools.ietf.org/html/rfc4122`.
getTenant()	The tenant owner of this record.
getType()	The type of this PO.
getValueAsBoolean()	Retrieve the value for the attribute with this name as a three-valued boolean (null, true or false).
getValueAsNumber()	Retrieve the value for the attribute with this name as a number.
getValueAsPO()	Retrieve the value for the attribute with this name as a PO or child of PO.
getValueAsString()	Retrieve the value for the attribute with this name as a string.
getValueAsTimestamp()	Retrieve the value for the attribute with this name as a timestamp in UTC and at millisecond precision.
persist()	Persist the changes to this instance.
setBoolean()	Set the boolean value for the attribute with this name.
setNumber()	Set the number value for the attribute with this name.
setPO()	Set the PO value for the attribute with this name.
setString()	Set the string value for the attribute with this name.
setTimestamp()	Set the timestamp value for the attribute with this name.

Table 3. The contract for the interface PO

Tenant Interface (extends PO)	
getName()	A textual identifier for this tenant.
isModule()	Whether this tenant is a module. A module only contains metadata, so no master data and no transaction data.
setName()	Set the name.
setModule()	Set whether this tenant is a module.

Table 4. The contract for the interface Tenant

User Interface (extends PO)	
getEmail()	An identifier for the user as an e-mail address as specified by RFC822: `http://www.freesoft.org/CIE/RFC/822/`.
getName()	Display name for this user.
setEmail()	Set the email. Must throw an exception if email is not a valid email address according to RFC822.
setName()	Set the name.

Table 5. The contract for the interface User

Type Interface (extends PO)	
getName()	A textual identifier for this Type. Must be unique within a tenant.
getAttributes()	Get all the attributes for this type in the current context.
getAttribute()	Get a single attribute for this type in the current context by name.
getAttribute()	Get a single attribute for this type in the current context by id.
setName()	Set the name for this type.
addAttributes()	Add one or more attributes to this type.

Table 6. The contract for the interface Type

Attribute Interface (extends PO)	
getDataType()	The data type of this attribute.
getMasterType()	The type this attribute belongs to.
getName()	A textual identifier for this Attribute. Must be unique within the master type.
isSearchable()	Whether this attribute is searchable. If this is true, the attribute can be used in search predicates.
setDataType()	Set the data type.
setSearchable()	Set whether this attribute should be searchable.
setMasterType()	Set the master type.

Table 7. The contract for the interface Attribute

		TINY	SMALL	MEDIUM
DT	Data tenants	10	100	1000
CF	Concurrency Factor	1	5	10
MDT	Master Data Types/Tenant	20	100	100
TDT	Transaction Data Types/Tenant	80	400	400
MDT	Master Data Types Instances/MDT/Tenant	2	2	2
STI	Search Type Instances	10,000	100,000	1,000,000
TI	Time Interval	60	300	300
MINRA	Minimum Reference Attributes on TDT	2	2	2
MAXRA	Maximum Reference Attributes on TDT	15	15	15

Table 8. Parameter Profiles

Profile The profile needs to be one of the options shown in Table 8. The profile *Tiny* is mostly meant for development purposes, to have a benchmark profile available that runs with minimal resources and allows for a quick test. The profile *Small* is for scenarios in which the system is expected to only accommodate a small number of tenants. The profile *Medium* should be a realistic scenario for many applications that are in need of an MTC-DB layer. We purposely omitted terms such as *Large* and *Very Large* to allow these terms to be added in the future, because it is to be expected that computer systems will keep scaling up as the available computing power and storage space keep growing exponentially.

The parameters mentioned in Table 8 have the following meanings:

1. **DT:** Data Tenants. The number of data tenants that are created in the setup script.
2. **CF:** Concurrency Factor. The number of threads each benchmark operation of the main script will use. Since there are 7 operations, the total number of concurrent threads the main script uses is 7 times *CF*.
3. **MDT**: Master Data Types. The number of master data types that are defined in the metadata module. We define master data as data that does not refer other data, but is referred to by transaction data.
4. **TDT**: Transaction Data Types. The number of transaction data types that are defined in the metadata module. We define transaction data as data that refers to *MINRA* to *MAXRA* (see below) master data records.
5. **MDI**: Master Data Type Instances. The number of master data type instances per tenant per type that will be created in the setup step of the test script.
6. **STI**: Search Type Instances. The number of search type instances. Used for benchmarking the search (see section 3.2).
7. **TI**: Test Interval. The duration of the test in seconds.
8. **MINRA**. Minimum Reference Attributes. The minimum number of reference attributes on transaction data types.

9. **MAXRA**. Maximum Reference Attributes. The maximum number of reference attributes on transaction data types.

Aulbach script The Aulbach script checks if the implementation is a correct MTC-DB implementation by testing if it is capable of representing the example MTC data structure used in the paper by Aulbach et al.[12]. This example consists of an Account table that is used by three tenants. One tenant uses the table with an extension for the health care industry, one with an extension for the automotive industry and one uses it without an extension.

Setup script The setup script sets up the MTC-DB implementation for running the main script. To this end, it creates several synthetic tenants, users, types and attributes. The amount of data it generates is heavily dependent on the chosen profile (see Table 8). The following actions are performed in sequential order.

1. Call `initializeSystem()`.
2. Call `createTenant()` to create a metadata tenant with the name "MainModule".
3. Call `createUser()` to create a user for the metadata module with the name "admin".
4. Create *MDT* number of master data types in this metadata module. For the *name* attribute, use the string "MDT" concatenated with a sequential number starting with 1.
5. Create *TDT* number of transaction data types in this metadata module. Each transaction data type has a random number of *MINRA* to *MAXRA* reference attributes to randomly picked master data types. For the *name* attribute, use the string "TDT" concatenated with a sequential number starting with 1.
6. Create a dedicated search type in the metadata module that contains 5 number attributes for testing the conjunctive search and 5 number attributes for testing the disjunctive search (see section 3.2).
7. Create *DT* tenants with `createTenant()` and create a user for each tenant with `createUser()`. Also add the metadata tenant to each tenant using `addModule()`.
8. Create a dedicated search tenant and instantiate *STI* instances of the search type in it. The number attributes for the conjunctive search are populated with random integers in the range $[1, \sqrt[5]{n}]$ and the number attributes for the disjunctive search are populated with random integers in the range $[1, 5n]$ (See section 3.2 for an explanation).
9. Instantiate *MDI* instances of master data types for each tenant.
10. Report the size on disk of the MTC-DB.

Main script The main script consists of seven operations that concurrently run for *TI* seconds. Each operation runs in *CF* concurrent threads, so the total number of threads is 7 times *CF*. The operations are:

1. **Create Tenants.** Every 5 seconds, create a new tenant with a random name and a dependency on the module "MainModule". Report the total number of tenants created and if it managed to achieve maximum performance.
2. **Create Types.** Every 500 milliseconds, create a new type with a random name for a random tenant. Report the total number of types created and if it managed to achieve maximum performance.
3. **Create Attributes.** Every 100 milliseconds, create a new searchable string attribute on with a random name for a random transaction data type for a random tenant. Reports the number of attributes created and if it managed to achieve maximum performance.
4. **Create Transaction Data Type Instances.** Constantly create transaction data type instances. Because these have references to *MINRA* to *MAXRA* master data types, this workload also includes retrieving master data types by name. Report the total number of TDI's created.
5. **Load by ID.** Constantly load previously created transaction data type instances by ID. This workload also includes loading all the references of the TDT to MDTs by ID. Reports the number of TDTs loaded.
6. **Conjunctive Search.** Constantly perform 5-way conjunctive (AND) searches. Only the first result is retrieved. It is designed in such a way that each search has about 63% chance of returning a result. Reports the number of conjunctive searches performed per minute. See section 3.2 for more information.
7. **Disjunctive Search.** Constantly perform 5-way disjunctive (OR) searches. Only the first result is retrieved. It is designed in such a way that each search has about 63% chance of returning a result. Reports the number of disjunctive searches performed per minute. See section 3.2 for more information.

Metrics The benchmark produces the following metrics:

1. **Aulbach compliance**: a boolean indicating whether the implementation is able to represent the example MTC scenario described by Aulbach et al. [12]. Every implementation needs to score *true* on this metric.
2. **Size on disk**: the total size on disk in MB of the MTC-DB after running the setup script. There is no maximum indication. The lower the better.
3. **Tenants created**: the percentage of tenants created in relation to the maximum possible amount. This should be 100%.
4. **Types created**: the percentage of types created in relation to the maximum possible amount. This should be 100%.
5. **Attributes created**: the percentage of attributes created in relation to the maximum possible amount. This should be 100%.
6. **TDI created per minute**: the number of transaction data type instances created per minute while running the main script. There is no minimum indication. The higher the better.
7. **TDI loaded by ID per minute**: the number of transaction data type instances loaded by ID per minute. There is no minimum indication. The higher the better.

8. **Conjunctive searches per minute**: the number of conjunctive searches performed per minute. There is no minimum indication. The higher the better.
9. **Disjunctive searches per minute**: the number of conjunctive searches performed per minute. There is no minimum indication. The higher the better.

3.4 Developer Guide

To help developers utilizing this benchmark with minimal effort, we implemented an example implementation in Java 8. This code is available under the MIT Licence at: `https://bitbucket.org/actfact/mtcdb-benchmark`. Developers can clone this Git repository and follow the instructions in the *readme* file. To implement their own MTC-DB implementation, they will need to write implementations for all the Java interfaces in the MTCB codebase. They are encouraged to refer to the example MTC-DB implementation or even use it as a starting point if they are unsure how to proceed. Of course it is also possible to write a non-Java implementation, but in this case the developer will have to first implement the API himself.

4 Evaluation

The evaluation consists of two parts. First we perform a conceptual evaluation, in which we evaluate if this benchmark fulfills the requirements we formulated in section 3.1. Second, we perform a practical evaluation. In this part, we discuss an MTC-DB example implementation we developed and how we used it to evaluate the usability of the benchmark.

4.1 Conceptual Evaluation

Realistic We defined a main module that contains the metadata for types that all data tenants use. In a real world situation it will also be the case that a large majority of metadata is the same for each tenant.

In the main script, concurrent users create new data, while other threads concurrently perform metadata operations. Metadata changes are a small part of the total workload of such applications, but it is important that regular data creation is not blocked while these operations are being performed. It was shown by Wevers that this is a significant problem for many Relational Databases [13].

Unambiguous We provide an implementation neutral specification and accompany this with an example implementation in Java 8. So wherever the specification leaves room for multiple interpretations, the example implementation can be referred to.

Comparable The benchmark specifies a small set of parameter profiles and produces a small number of simple quantitative metrics. This enables easy and objective comparison of different implementations that use the same profile.

Correct The *Aulbach script* is a minimal test that checks if the implementation is a real MTC-DB implementation. Currently no automated check is implemented for ACID compliance. On a more general note, it is not possible to automatically guarantee complete correctness. We can only check if the implementation is consistent in itself. To guarantee correctness an audit by a human expert will always remain necessary.

Scalable The parameter profiles allow for benchmarking a number of scenarios of different sizes.

Simple Instead of specifying a real world data model for the main module, we chose to use synthetic types and attributes. The same goes for the scripts that generate data and metadata: it is randomized and without meaning. Using a real world scenario would make MTCB extremely complex and would decrease its scalability and flexibility.

Independent We specify an API that contains the operations that should be supported by the MTC-DB implementation. This API places no restrictions on the MTC-DB implementation in terms of underlying platform. For example, even though many implementations will use an RDBMS as underlying platform, this is not implied in the API. It should be equally possible to implement the MTC-DB in a document-oriented database, a functional database or any other kind of persistent storage structure.

4.2 Practical Evaluation

For the practical evaluation, we developed a naive MTC-DB implementation. This implementation was developed in Java 8 and PostgreSQL 9.6 and is schema based: every tenant is defined in a separate schema. It is loosely based on what Aulbach et al. call the Private Table Layout [12]. It is available on the same repository as the benchmark itself, in the project mtcb-schemabased: `https://bitbucket.org/actfact/mtcdb-benchmark`.

We ran MTCB for this implementation on a Centos 7 server with an Intel Xeon E3-2200 QuadCore CPU and 32 GB RAM. For each profile we ran the main script 10 times and report the average (μ) as well as the coefficient of variation ($\frac{\sigma}{\mu}$) in Table 9. The reason to run it 10 times was that we noticed considerable differences between separate runs. This can be seen from the high variation for some metrics. We did not benchmark this implementation for the medium profile, because it seems to not be feasible for such a large scenario. We estimate that running the setup script would not even finish in 48 hours.

	Tiny		Small	
Aulbach compliance	True		True	
Size on disk	138 MB		5,471 MB	
	μ	$(\frac{\sigma}{\mu})$	μ	$(\frac{\sigma}{\mu})$
Tenants created	40%	(0.09)	5%	(0.00)
Types created	100%	(0.00)	100%	(0.00)
Attributes created	23%	(0.51)	67%	(0.29)
TDI created per minute	1,504	(0.47)	2,442	(0.17)
TDI loaded by ID per minute	144,585	(0.29)	168,722	(0.21)
Conjunctive searches per minute	85,461	(0.02)	9,074	(0.06)
Disjunctive searches per minute	665,173	(0.04)	580,297	(0.06)

Table 9. The benchmark result over 10 runs for a schema based implementation, showing the average (μ) and the coefficient of variation ($\frac{\sigma}{\mu}$).

This implementation scores very well on some metrics. Most notably the disjunctive search: more than half a million per minute for both profiles. Loading TDIs is also fast.

On some of the other metrics the implementation scores very poorly. The largest problem is tenant creation. The implementation fails to comply with the requirement to create a new tenant every 5 seconds. For the small scenario, it only creates 5% of the maximum. This means it takes about 100 seconds to create a tenant. The reason for this is that in this implementation, for each tenant creation the database must run DDL[8] to create all the tables that are defined in the metadata module. Aside from taking a lot of time, this also causes the implementation to score poorly on the metric *Size on disk*. On top of this, the DDL statements have a disruptive nature, irregularly causing operations such as TDI Creation to be stalled for considerable times. This causes a high variation for those operations.

Another interesting note is that the performance does not degrade much when going from the tiny to the small profile. For some metrics, the performance even increases. The most likely reason for this is that the medium profile has a higher degree of concurrency, running in 35 threads, whereas the tiny profile only runs in 7 threads. This allows the medium profile to maximize its use of the hardware resources. However, the conjunctive search still degrades severely. This is probably due to the stark increase in search data: 10 times as much in the small profile.

[8]Data Definition Language. SQL statements that alter the data dictionary: mostly `CREATE TABLE` and `ALTER TABLE` statements

5 Conclusion

These results prove that *MTCB* is implementable. They also show that a naive schema per tenant RDBMS implementation is not sufficient, because it cannot handle metadata modifications efficiently and causes a huge overhead in redundant metadata storage. Future work should use this example implementation as a baseline system. An interesting next step would be to create implementations based on the schema-mapping techniques discussed by Aulbach et al. [12].

We believe that this benchmark is an important contribution to the community of MTC-DB developers. Not only does it allow for objective comparison, it also makes an attempt at a very precise definition of the concept of MTC-DB, backed by a concrete implementation.

References

1. R. M. Locke, "The promise and perils of globalization: The case of nike.," *MIT Working Paper*, 2002. Downloaded 16 December 2016 from https://ipc.mit.edu/sites/default/files/documents/02-007.pdf.
2. W. R. Friedrich and J. A. Van Der Poll, "Towards a methodology to elicit tacit domain knowledge from users," *Interdisciplinary Journal of Information, Knowledge, and Management*, vol. 2, pp. 179–193, 2007.
3. R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014.
4. R. Krebs, A. Wert, and S. Kounev, *Multi-tenancy performance benchmark for web application platforms*, vol. 7977 LNCS of *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2013.
5. K. Shanley, "History and overview of the tpc," 1998. Accessed 16 December 2016 on http://www.tpc.org/information/about/history.asp.
6. Transaction Processing Performance Council, "TPC BENCHMARK C Standard Specification Revision 5.11," 2010.
7. Transaction Processing Performance Council, "TPC BENCHMARK E Standard Specification Version 1.14.0," 2015.
8. S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica, "Tpc-e vs. tpc-c: Characterizing the new tpc-e benchmark via an i/o comparison study," *SIGMOD Record*, vol. 39, no. 3, pp. 5–10, 2010.
9. W. D. Smith, "Tpc-w: Benchmarking an ecommerce solution," 2000. Accessed 16 December 2016 on `http://www.tpc.org/tpcw/tpc-w_wh.pdf`.
10. salesforce.com, "The force.com multitenant architecture: Understanding the design of salesforce.com's internet application development platform," 2008. Accessed 4 January 2017 on `http://www.developerforce.com/media/ForcedotcomBookLibrary/Force.com_Multitenancy_WP_101508.pdf`.
11. C.-P. Bezemer and A. Zaidman, "Challenges of reengineering into multi-tenant saas applications," *Delft University of Technology Software Engineering Research Group. Technical Report Series*, 2010.
12. S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, "Multi-tenant databases for software as a service: Schema-mapping techniques," in *SIGMOD '08. Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1195–1206, ACM, 2008.

13. L. Wevers, “A persistent functional language for concurrent transaction processing,” Master’s thesis, University of Twente, 2012.

Evaluating Multi-Tenant Customizable Database Implementations

Wim van der Zijden

University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands
ActFact Projects, Josink Esweg 8, 7545 PN Enschede,The Netherlands
w.vanderzijden@actfact.com

Abstract. For most companies IT activities distract from their main activities and should therefore be outsourced whenever possible. However, many of those companies need highly customized OLTP software. This need is fulfilled by Multi-Tenant Customizable OLTP. To build such a platform there is a need for Multi-Tenant Customizable Databases (MTC-DB). We designed four MTC-DB implementations on top of a traditional relational database: two Field Based approaches and two Row Based approaches. We compared these approaches using a previously designed MTC-DB benchmark. This benchmark supplies an example implementation of the benchmark and a baseline MTC-DB implementation. The test results show that each implementation has some metrics that it scores best on. On some metrics the Schema Based baseline system is fastest, namely the speed of conjunctive and disjunctive searches. We conclude that future work should try to combine the four implementations and develop a smart query planner that can decide which execution method is best for each individual query.

Keywords: Multi-Tenant Customizable, Multi-level customizability, OLTP, PaaS, Database, Benchmark

1 Introduction

For most companies IT activities distract from their main activities and should therefore be outsourced whenever possible. However, most existing cloud solutions are not sufficient, because many companies are in need of highly customized OLTP[1] software. A solution for this predicament is Multi-Tenant Customizable (MTC):

> *A single software solution that can be used by multiple organizations at the same time and which is highly customizable for each organization and user within that organization, by domain experts without a technical background.* [1]

[1]Online Transaction Processing. Applications whose main concern it is to persistently and reliably process live data transactions and to facilitate reporting on this data. An OLTP application is usually backed by a relational database.

This is especially interesting from a vendor point of view, because in comparison to multiple single tenant solutions, it is much easier to deploy new tenants, to update all tenants at once and to scale up.

These kind of applications need an MTC Database (MTC-DB) as a storage structure. In this paper we discuss four different MTC-DB implementations and use the benchmark *MTCB* that we introduced in previous work [1] to evaluate them. To this end, we formulate the following technical research problem, using the template proposed by Wieringa [2].

> ***Improve*** *the development of Multi-Tenant Customizable OLTP (MTC-OLTP),* ***by developing*** *a Multi-Tenant Customizable Database (MTC-DB)* ***such that*** *it is compliant with the benchmark MTCB [1] and scores significantly higher on the benchmark's metrics than the baseline system* ***in order to*** *enable developers to use this MTC-DB to develop MTC-OLTP platforms.*

In previous work [1] we provide a more extensive motivation and description of MTC-OLTP, as well as a problem analysis of existing on-premises OLTP and a description of the application requirements for such systems.

2 Literature Review

2.1 MTCB: A Multi-Tenant Customizable DB Benchmark

In previous work we developed a benchmark for MTC-DBs called *MTCB*[1]. This benchmark specifies six interfaces that an MTC-DB should implement, and defines nine metrics that are to be reported. Furthermore, it specifies three parameter profiles: tiny, small and medium. The use of these profiles allows for easy comparison of MTC-DB implementations. See Table 1 for an overview of these parameter profiles. Aside from the benchmark specification, there is an example implementation available in Java 8 and a compliant MTC-DB implementation in Java 8 and PostgreSQL 9.6[2].

This baseline MTC-DB implementation is called schema based and is similar to what Aulbach et al. [3] call the Private Table Layout. The evaluation of this implementation shows that it is fast in search queries, but very slow in metadata operations, especially creating tenants, and also has a very large overhead in size on disk because of redundancy of metadata in PostgreSQL's data dictionary. Being able to create tenants quickly and without much overhead is important, because an MTC-OLTP should ideally offer a one click registration process that makes it easy to try out the application.

This benchmark has a very pure view on what an MTC-DB is: it must be fully customizable. This means that an approach such as the Extension Table Layout as discussed by Aulbach et al. [3] is not sufficient, because this approach

[2]The example implementations are available on `https://bitbucket.org/actfact/mtcdb-benchmark`

		TINY	SMALL	MEDIUM
DT	Data tenants	10	100	1000
CF	Concurrency Factor	1	5	10
MDT	Master Data Types/Tenant	20	100	100
TDT	Transaction Data Types/Tenant	80	400	400
MDT	Master Data Types Instances/MDT/Tenant	2	2	2
STI	Search Type Instances	10,000	100,000	1,000,000
TI	Time Interval	60	300	300
MINRA	Minimum Reference Attributes on TDT	2	2	2
MAXRA	Maximum Reference Attributes on TDT	15	15	15

Table 1. Parameter Profiles in the MTC-DB Benchmark [1]

assumes that a large part of the data model is the same for each tenant. In this benchmark only very basic metadata concepts are part of the core data model: the primitive types string, number, timestamp and boolean and the complex types tenant, user, type and attribute.

2.2 Relational Databases

The Relational Database Management System (RDBMS) is by far the most commonly used type of database in OLTP applications, and in fact, in any application that is in need of persistent storage. Because of its widespread use we will assume that the reader is familiar with relational databases, including their design using Entity-Relationship Diagrams (ERD), the Structured Query Language (SQL) that is used to manipulate them and the concept and purpose of data normalization.

Despite their widespread use, RDBMS's do not have native support for multi-tenant customizability. In this section we will discuss how they can still be used in such an environment.

Three Approaches to MTC-DBs Jacobs et al. [4] introduce three fundamentally different approaches to using a relational database as an MTC-DB.

- **Shared machine:** each tenant has its own database process, and multiple tenants share the same machine. A strong advantage of this approach is isolation between tenants, complemented with the disadvantage that sharing data is complex. A major downside is that it requires massive duplication of metadata of the core functionality of the platform.
- **Shared Process:** tenants share the same database process, but get their own tables. This approach is favoured by the authors. A disadvantage that the authors fail to mention is that this also requires duplication of the core

platform metadata in the DBMS's data dictionary. It also forces the DBMS to duplicate the query cache for each tenant.
- **Shared Table:** applications share tables. It scales better than shared process and is also more efficient in pooling resources. The major problem of this approach is that a lot of concerns that are normally handled by the DBMS, must now be handled in the application. For example: access rights, query optimization and constraint enforcement.

Schema-Mapping Techniques Aulbach et al. [3] describe seven ways to provide mappings for single-tenant schemas to multi-tenant schemas. These are described below. Figure 1 illustrates these techniques by showing the appropriate data structure for these techniques using a simple example. In this example there is an Account table that is used by three tenants: 17, 35 and 42. Tenant 17 has an extension for the health care industry and tenant 42 has an extension for the automotive industry. The Basic Layout is not illustrated, because it cannot express this scenario because it does not support multi-tenant customizability. Figure 1 only shows the data tables. Additional tables are needed to store metadata like field names.

- **Basic Layout.** Each table has a tenant ID column and tables are shared among tenants. This approach does not provide multi-tenant customizability - instead, tenants must share customizations.
- **Private Table layout.** Each tenant has their own tables. This works quite well for a limited amount of tenants. When there are many tenants, the Data Dictionary of the database may grow too large to handle.
- **Extension Table layout:** A combination of the Basic and Private Table layout. Tenants share tables, but extension tables are used for custom fields. Extension tables may also be shared. This setup leads to less tables than in the Private Table Layout, but the number of tables still grows proportionally with the number of tenants.
- **Universal table layout.** This is a layout where one generic table is used to store arbitrary entities. It has a tenant column, a table column, and a set number of generic data columns (eg. 250). This approach has its origin in a 1983 paper by Maier and Ullman [5]. The main advantage of this approach is that no DDL[3] is required for adding new customizations. A downside is that these generic columns do not allow for indexing. Another downside is that the rows in this table will contain a large amount of null values. The DBMS must be able to handle this efficiently.
- **Pivot table layout.** The value of each field is stored in a separate row in a pivot table. For each data type there is a pivot table, eg: Pivot_int, Pivot_str, etc. The advantage over the Universal Table Layout is that it does not circumvent typing. Therefore, meaningful indexes can be created on these tables.

[3]Data Definition Language. SQL Statements that alter the data dictionary such as `CREATE TABLE` and `ALTER TABLE`.

Account$_{17}$

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

Account$_{35}$

Aid	Name
1	*Ball*

Account$_{42}$

Aid	Name	Dealers
1	*Big*	65

(a) Private Table Layout

Account$_{Ext}$

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

Healthcare$_{Account}$

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

Automotive$_{Account}$

Tenant	Row	Dealers
42	0	65

(b) Extension Table Layout

Universal

Tenant	Table	Col1	Col2	Col3	Col4	Col5	Col6
17	0	*1*	*Acme*	*St. Mary*	*135*	–	–
17	0	*2*	*Gump*	*State*	*1042*	–	–
35	1	*1*	*Ball*	–	–	–	–
42	2	*1*	*Big*	*65*	–	–	–

(c) Universal Table Layout

Pivot$_{Int}$

Tenant	Table	Col	Row	Int
17	0	0	0	1
17	0	3	0	135
17	0	0	1	2
17	0	3	1	1042
35	1	0	0	1
42	2	0	0	1
42	2	2	0	65

Pivot$_{Str}$

Tenant	Table	Col	Row	Str
17	0	1	0	*Acme*
17	0	2	0	*St. Mary*
17	0	1	1	*Gump*
17	0	2	1	*State*
35	1	1	0	*Ball*
42	2	1	0	*Big*

(d) Pivot Table Layout

Chunk$_{Int|Str}$

Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	1	*Acme*
17	0	1	0	135	*St. Mary*
17	0	0	1	2	*Gump*
17	0	1	1	1042	*State*
35	1	0	0	1	*Ball*
42	2	0	0	1	*Big*
42	2	1	0	65	–

(e) Chunk Table Layout

Account$_{Row}$

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

Chunk$_{Row}$

Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	135	*St. Mary*
17	0	0	1	1042	*State*
42	2	0	0	65	–

(f) Chunk Folding

Fig. 1. Schema mapping techniques by Aulbach et. al.[3] Gray columns indicate meta-data.

- **Chunk Table Layout.** This layout combines multiple pivot tables into one table. The values of multiple fields are stored in one row. These groups of fields are called chunks. This approach has all the benefits of the Pivot Table Layout, but reduces the number of rows needed. The price to pay for this is that the query-transformation logic becomes more complicated.
- **Chunk folding.** This approach combines the Extension Layout with the Chunk Table Layout. The application's core entities are represented with conventional tables and the extensions are modelled in chunk tables, one chunk table for each conventional table.

Of these techniques, the Chunk Table Layout is proposed by the authors. Its advantage over the Pivot Table Layout is a reduction of metadata overhead. Another advantage is that the logical entity can be constructed from less rows. The advantage over the Universal Table Approach is that it automatically incorporates a structure to add indexes. A clear disadvantage of this approach is that the query transformation is much more complex, because the chunks need to be resolved.

The Chunk Folding technique is also proposed by the author and is their method of choice. Its main advantage over the Chunk Table Layout is that often it may not be necessary to join the extension tables. Therefore, a lot of the times the performance may be similar to a Basic Table Layout structure. Its main advantage over the Universal Table Layout is that the tables can contain much less columns.

An important characteristic of these techniques is that the size of the chunks can be manipulated with. This way, a middle ground can found between the Universal Table Layout and the Pivot Table Layout.

2.3 Document Databases

An interesting alternative to a relational database are document databases such as MongoDB [6]. The characteristics of these databases is that they do not require DDL and that they are optimized for navigational access through embedded document structures. However, a central design principle of these databases is to store data unnormalized. This approach seems unsuitable for the problem at hand, because we in fact need to further normalize our data. For a regular relational database, the data is normalized for a single tenant. In this research, we are looking for an effective way to normalize over multiple tenants, i.e. remove the data redundancy that exists over multiple tenants. Because documents databases move in the opposite direction, this makes them unsuitable by definition. However, as a secondary storage they will probably be very interesting. For example, as an "eventually consistent" database that is optimized for full text search.

2.4 Graph Databases

A new upcoming paradigm in databases is the graph database, such as Neo4j [7]. In a graph database, the relations between entities are first class citizens, as

opposed to relational databases, where the entities are first class citizens. This makes graph databases very promising for business applications, because it is much faster to navigate through a path, than in SQL, where each navigation to a related table requires an expensive join-operation.

However, we were not able to find any previous research into multi-tenant setups with graph databases. Another problem is that graph databases are much less well-studied than relational databases, and therefore it is hard to know how the internals work exactly, and what kind of setups will increase or decrease performance. We can also assume that the stability will be lower than with one of the major RDBMS's.

2.5 Functional Databases

The functional database, sometimes referred to as a persistent functional programming language, is yet another alternative for the Relational Database. Extensive research has been done in this topic, for example by Wevers [8]. However, currently there is no production ready implementation of this concept in sight, which makes it unsuitable for our purposes.

2.6 Column Based Databases

There also exist relational databases that break with the traditional row-based paradigm, and implement a column based approach instead. This seems similar to what Aulbach et al. [3] call a Pivot Table Layout. The most notable implementation of such a database is MonetDB as described by Manegold et al. [9]. However, this database does not seem suitable for the following reasons:

1. Even though it has been in development for over 20 years, it seems stuck in an academic, experimental phase. It does not seem to be used in production environments very much. We have not been able to find examples of enterprise production systems using MonetDB. Another sign that it is hardly in use is the lack of questions on the popular programming Q&A site Stack Overflow: only about 275 questions with the tag 'monetdb'. As a comparison: there are about 12,000 questions with the tag 'neo4j', and around 63,000 questions with the tag 'postgresql'[4].
2. MonetDB does not natively support multi-tenancy. It is an alternative implementation of an RDBMS and as such does not necessarily solve any of the problems discussed here.

3 Approaches

For designing our own approaches we used the benchmark discussed in section 2.1. We reused the provided Java 8 API and developed four implementations,

[4]These numbers were based on information on `http://stackoverflow.com/tags` on 7 January 2017

based on two distinct approaches, both based on an RDBMS. The reasons we chose a classic SQL-based RDBMS are:

1. **Stability:** SQL databases have been the number one choice for enterprise applications and have proven to be very stable.
2. **Well-known:** the mechanics, architecture and theory behind SQL Databases is well-known and well-documented.
3. **Lack of viable alternatives.** As this point in time, there simply do not seem to be many viable alternatives.

The two approaches are based on the techniques discussed in 2.2. We call them the Field Based and Row Based approaches. They are based respectively on the Universal Table Layout and the Pivot Table Layout. We chose these methods because these are the most fundamental extremes. For each approach we also developed an alternative implementation that differ in the way the conjunctive and disjunctive search search were implemented.

One shortcoming of the description of the approaches discussed by Aulbach et al. is that the metadata model is not considered. The metadata model contains data such as field names. In our model we design the metadata model in detail, partly because it is required by the MTC-DB Benchmark that we are using, but also because the metadata model is an essential part of any MTC-DB. For example, it would be impossible to account for modularity without including the metadata model in the implementation. Modularity is necessary to allow metadata to be shared over multiple tenants, which is a crucial part of an MTC-DB.

3.1 Field Based Approach

Because the Row Based Approach is built as an extension of the Field Based Approach, we first discuss the Field Based Approach and then discuss the differences in the Row Based Approach. The Field Based Approach is similar to the Pivot Table Layout as discussed in 2.2. The main difference is that this model also accounts for modularity - metadata can be reused over multiple tenants. Another difference is that instead of having one table for each database datatype, it uses just one table for all data. This is cleaner and makes little difference in performance, because we can create indexes that ignore `NULL` values.

The ERD[5] for the Field Based Approach is shown in figure 2. Each entity of this diagram is discussed below.

Tenant The basic meaning of a tenant is something like "an organizational entity which rents a multi-tenant SaaS solution." [10] In the MTC-DB Benchmark, this definition of tenant has been extended. To allow for modularity and shareable metadata over multiple tenants, a distinction is made between data tenants and module tenants. A data tenant is a tenant that holds isolated master

[5] Entity Relationship Diagram. A common technique to model relational databases.

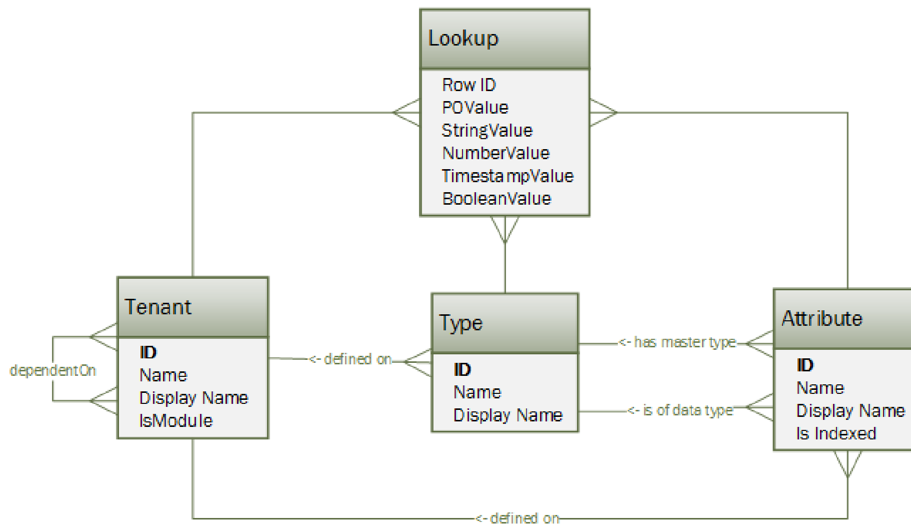

Fig. 2. ERD for the Field Based Approach.

and transaction data for one organization, and little to no metadata. A module tenant is a tenant that only holds shared metadata that can be used by data tenants and other module tenants.

Each row in the database references a single tenant, the owner of the record. Tenants may be dependent on other tenants, as defined by the recursive many-to-many relationship in Figure 2.

Type Table and Attribute Table The Type Table is used to define the types in the system. The Attribute Table is used to define attributes on these types. Types have 0 or more attributes. A type with 0 attributes is a simple type and a type with multiple attributes is a complex type. An example of a simple type is Number. An example of a complex type is Account, as used in the running example illustrated in Figure 1. Each attribute has two references to a type: the type that it belongs to, and its datatype. The datatype will often be a primitve type, but may be a complex type as well. For example, let's say we have the type Order. Order may have the following attributes:

1. DocNumber: datatype String
2. Customer: datatype Customer
3. OrderDate: datatype Timestamp
4. GrandTotal: datatype Number
5. IsShipped: datatype Boolean

The Attribute Table also has a reference to the Tenant in which it was defined. This is needed because an attribute may be defined in a different tenant than the tenant that defined its type.

Lookup Table The Lookup Table stores all the actual tenant data, in other words, the instantiations of complex types. It uses one row per field and has a separate column for each data type: PO, Text, String, Number, Timestamp and Boolean. PO stands for persistent object and is a reference to the instantiation of another complex type. The others are primitive types. In a particular row, only one of these columns will hold a value, the others will be empty. For each of these columns an index is maintained, e.g.: (Attribute, POValue, Row), (Attribute, StringValue, Row), etc. Row is included in these indexes to enable the query optimizer to do an "index only scan" - it can use just the index to get the row id, and does not need to access the Lookup Table itself. Finally, it also has an index on (Attribute, Row). This index is needed for updating values.

Considerations An important aspect of this approach is that every single field is indexed. This could have a big adverse impact on the performance of inserting and updating data. If this is the case we could extend the model with a second Lookup Table without indexes on the value columns and use this table to store the values for attributes that do not have to be indexed.

Tenant		
ID	**Name**	**Category**
0	System	System
1	Finance	Metadata
2	Health Care	Metadata
3	Automotive	Metadata
17	Hospital X	Tenant
35	Bank X	Tenant
42	Garage X	Tenant

Dependent Tenant	
Tenant	**DependentTenant**
1 (Finance)	0 (System)
2 (Health Care)	1 (Finance)
3 (Automotive)	1 (Finance)
17 (Hospital X)	2 (Health Care)
35 (Bank X)	1 (Finance)
42 (Garage X)	3 (Automotive)

Type		
ID	**Tenant**	**Name**
0	0 (System)	String
1	0 (System)	Integer
2	1 (Finance)	Account

Attribute				
ID	**Tenant**	**MasterType**	**DataType**	**Name**
0	1 (Finance)	2 (Account)	0 (String)	Name
1	2 (Health Care)	2 (Account)	0 (String)	Hospital
2	2 (Health Care)	2 (Account)	1 (Integer)	Beds
3	3 (Automotive)	2 (Account)	1 (Integer)	Dealers

Fig. 3. Example data for the metadata tables using the running example from Figure 1.

Lookup					
Row	**Tenant**	**Type**	**Attribute**	**StringValue**	**NumberValue**
1	17 (Hospital X)	2 (Account)	0 (Name)	Acme	
1	17 (Hospital X)	2 (Account)	1 (Hospital)	St. Mary	
1	17 (Hospital X)	2 (Account)	2 (Beds)		135
2	17 (Hospital X)	2 (Account)	0 (Name)	Gump	
2	17 (Hospital X)	2 (Account)	1 (Hospital)	State	
2	17 (Hospital X)	2 (Account)	2 (Beds)		1042
3	35 (Bank)	2 (Account)	0 (Name)	Ball	
4	42 (Garage X)	2 (Account)	0 (Name)	Big	
4	42 (Garage X)	2 (Account)	3 (Dealers)		65

Fig. 4. Example data for Field Based approach using the running example from Figure 1. Read in combination with the metadata in Figure 3.

Example In Figure 3 and Figure 4 we show how the running example by Aulbach et al. from Figure 1 can be represented in this table structure. To increase the readability of this example, whenever an ID references a record in a different table, the name of that record is added in parenthesis. In the Lookup Table, some unused columns have been left out, such as TimestampValue and BooleanValue.

3.2 Row Based Approach

Because the Row Based Approach is an extension of the Field Based Approach, we will only discuss the differences with the Field Based Approach. Its ERD is shown in Figure 5.

PO Table The main difference is that we have an extra table: PO. PO stands for persistent object. This table stores all data for all tenants, using one row per instance. This is in contrast with the Lookup Table in the Field Based Approach, which uses one row for each field.

The PO Table has one column that refers to the tenant owner of the record, one column referring to the type that it is an instance of, and 250 generic variable length string columns, named `c0` through `c249`. These columns are used to store the attribute values. When storing data, the values are converted to strings, and when retrieving the data, the values are converted from strings back to their original datatype. The Attribute Table has an extra column `ColNo` that indicates which generic column is used to store the values for that attribute.

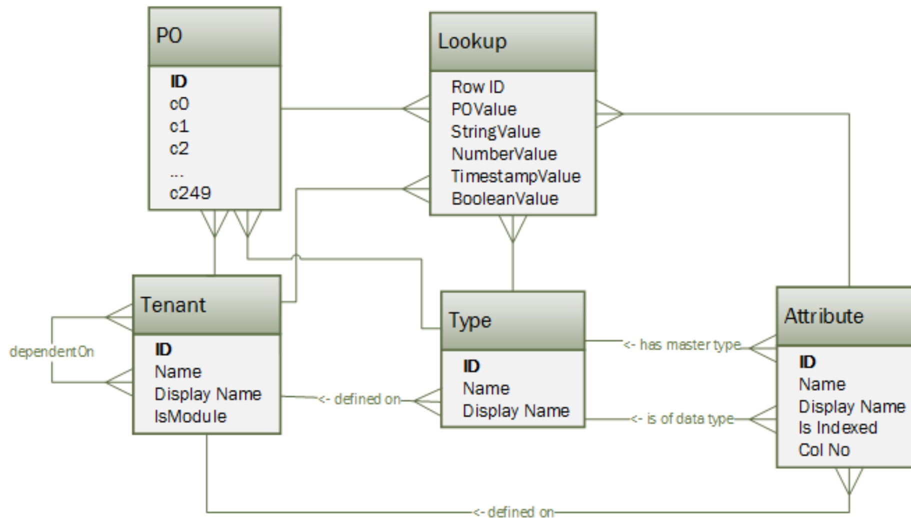

Fig. 5. ERD for the Row Based Approach

Despite using the PO Table to store all data, the Lookup Table from the Field Based Approach is still used as well, but only for storing values for indexed attributes.

Example In Figure 6 we show how this model can be used to represent the running example by Aulbach et al. from Figure 1. To increase the readability of this example, whenever an ID references a record in a different table, the name of that record is added in parenthesis. The Lookup Table has been left out, because it is only used for indexing in this model, so not needed to represent the example. This example should be understood in combination with the example of Figure 3.

3.3 Query Transformation

Based on these two approaches, we developed four implementations: Field Based, Field Based Alternative, Row Based and Row Based Alternative. In this section we will discuss how a number of typical OLTP queries were implemented in these implementations. For this we will use the running example from Figure 1. First we will show what the query would look like in the Private Table Layout, then we will show how these were implemented. The metadata is queried separately, but we will not discuss this, because this is very straightforward.

Creating a new instance

Create a new Account for Tenant 42 with name Small and Dealers 10

Attribute					
ID	**Module**	**MasterType**	**DataType**	**Name**	**ColNo**
0	1 (Finance)	2 (Account)	0 (String)	Name	0
1	2 (Health Care)	2 (Account)	0 (String)	Hospital	1
2	2 (Health Care)	2 (Account)	1 (Integer)	Beds	2
3	3 (Automotive)	2 (Account)	1 (Integer)	Dealers	3

PO						
ID	**Module**	**Type**	**c0**	**c1**	**c2**	**c3**
1	17 (Hospital X)	2 (Account)	Acme	St. Mary	135	
2	17 (Hospital X)	2 (Account)	Gump	State	1042	
3	35 (Bank)	2 (Account)	Ball			
4	42 (Garage X)	2 (Account)	Big			65

Fig. 6. Example data for Row Based approach using the running example from Figure 1. Read in combination with the metadata in Figure 3.

For this query type there is no difference in the alternative implementations. The Field Based Approach inserts one row into the Lookup Table for each non-null attribute. The Row Based Approach inserts one row into the PO Table, but also one row into the Lookup Table for each non-null indexed attribute.

Private Table Layout:

```
INSERT INTO T42_Account (ID, Name, Dealers)
VALUES (5, 'Small', 10)
```

Field Based and Field Based Alternative:

```
INSERT INTO Lookup (Tenant, Type, Attribute, Row,
    StringValue, NumberValue)
VALUES
(17, 2, 0, 5, 'Small', NULL),
-- 17=Hospital X, 2=Account, 0=Name
(17, 2, 3, 5, NULL, 10)
-- 17=Hospital X, 2=Account, 3=Beds
-- Only non-null attributes are inserted
```

Row Based and Row Based Alternative:

```
INSERT INTO PO (Tenant, Type, ID, c0, c3)
VALUES (17, 2, 5, 'Small', '10')
-- 17=Hospital X, 2=Account
-- Additional inserts in the Lookup Table are needed for each
    attribute that is indexed
```

Update existing instance

Update the number of beds to 150 for the Account with ID 1 in Tenant 17

For this query type, there is no difference in the alternative implementations. If the field is updated from non-null, the Field Based Approach performs the update statement below. If it is updated from null, then the insert statement from section 3.3 is used. If the field is updated from non-null to null, then a delete statement is used. The Row Based Approach updates the PO Table, but also needs a similar statement as the Field Based Approach when updating indexed attributes.

Private Table Layout:

```
UPDATE T17_Account
SET beds = 150
WHERE ID = 1
```

Field Based and Field Based Alternative:

```
UPDATE Lookup
SET NumberValue = 150
WHERE Row = 1
AND Attribute = 2 -- Beds
-- If the value is updated from null, this should be an
    insert.
```

Row Based and Row Based Alternative:

```
UPDATE PO
SET c1 = '150'
WHERE ID = 1
```

Disjunctive Search

Select all rows from Account for Tenant 17 where name is 'Acme' or beds equals 1042

This query was implemented differently in each implementation. In Field Based, multiple queries are combined with `UNION` to achieve the result. In Field Based Alternative, no `UNION`, but only `JOIN` is used. In both cases the query returns 1 row per field, ordered by row. The object is constructed in the application layer.

In Row Based a similar method as in Field Based is used, but it returns one row per instantiation. In Row Based Alternative the Lookup Table is not used, and instead the unindexed PO Table is queried directly.

Private Table Layout:

```
SELECT * FROM T17_Account
WHERE Name = 'Acme'
OR Beds = 1042
```

Field Based:

```
SELECT l.*
FROM Lookup l
JOIN Lookup l0
ON l0.Attribute = 0 -- Name
AND l0.StringValue = 'Acme'
AND l0.Row = l.Row
WHERE l.Tenant = 17 -- Hospital X
AND l.Type = 2 -- Account
UNION
SELECT l.*
FROM Lookup l
JOIN Lookup l0
ON l0.Attribute = 2 -- Beds
```

```
AND l0.NumberValue = 1042
AND l0.Row = l.Row
WHERE l.Tenant = 17 -- Hospital X
AND l.Type = 2 -- Account
ORDER BY Row
-- Returns a row for each non-null attribute in Account
```

Field Based Alternative:

```
SELECT l.*
FROM Lookup l
JOIN Lookup l0
ON l0.Attribute = 0 -- Name
AND l0.Row = l.Row
JOIN Lookup l1
ON l1.Attribute = 2 -- Beds
AND l1.Row = l.Row
WHERE l.Tenant = 17 -- Hospital X
AND l.Type = 2 -- Account
AND (l0.StringValue = 'Acme'
OR l1.NumberValue = 1042)
ORDER BY l.Row
-- Returns a row for each non-null attribute in Account
```

Row Based:

```
SELECT po.*
FROM PO
JOIN Lookup l0
ON l0.Attribute = 0 -- Name
AND l0.StringValue = 'Acme'
AND l0.Row = po.ID
WHERE po.Tenant = 17 -- Hospital X
AND po.Type = 2 -- Account
UNION
SELECT po.*
FROM PO
JOIN Lookup l0
ON l0.Attribute = 2 -- Beds
AND l0.NumberValue = 1042
AND l0.Row = po.ID
WHERE po.Tenant = 17 -- Hospital X
AND po.Type = 2 -- Account
```

Row Based Alternative:

```
SELECT *
FROM PO
WHERE Tenant = 17 -- Hospital X
```

```
AND Type = 2 -- Account
AND (c0 = 'Acme'
OR c2 = '1042')
```

Conjunctive Search

> *Select all rows from Account for Tenant 17 where name is 'Gump' and hospital is 'State'*

As with the disjunctive search, this query was implemented differently in each implementation. Field Based uses only `JOIN`, and Field Based Alternative uses `INTERSECT` to merge intermediate results. Row Based uses only `JOIN` as well, and Row Based Alternative skips the Lookup Table and queries the unindexed PO Table directly.

Private Table Layout:

```
SELECT *
FROM T17_Account
WHERE Name = 'Gump'
AND Hospital = 'State'
```

Field Based:

```
SELECT l.*
FROM Lookup l
JOIN Lookup l0
ON l0.Attribute = 0 -- Name
AND l0.Row = l.Row
JOIN Lookup l1
ON l1.Attribute = 1 -- Hospital
AND l1.Row = l.Row
WHERE l.Tenant = 17 -- Hospital X
AND l.Type = 2 -- Account
AND l0.StringValue = 'Gump'
AND l1.StringValue = 'State'
ORDER BY Row
-- Returns a row for each non-null attribute in Account
```

Field Based Alternative:

```
SELECT l.*
FROM Lookup l
JOIN Lookup l0
ON l0.Attribute = 0 -- Name
AND l0.StringValue = 'Gump'
AND l0.Row = l.Row
WHERE l.Tenant = 17 -- Hospital X
AND l.Type = 2 -- Account
```

```sql
INTERSECT
SELECT l.*
FROM Lookup l
JOIN Lookup l0
ON l0.Attribute = 1 -- Hospital
AND l0.StringValue = 'State'
AND l0.Row = l.Row
WHERE l.Tenant = 17 -- Hospital X
AND l.Type = 2 -- Account
ORDER BY Row
-- Returns a row for each non-null attribute in Account
```

Row Based:

```sql
SELECT l.*
FROM PO
JOIN Lookup l0
ON l0.Attribute = 0 -- Name
AND l0.Row = po.ID
JOIN Lookup l1
ON l1.Attribute = 1 -- Hospital
AND l1.Row = po.ID
WHERE po.Tenant = 17 -- Hospital X
AND po.Type = 2 -- Account
AND l0.StringValue = 'Gump'
AND l1.StringValue = 'State'
```

Row Based Alternative:

```sql
SELECT *
FROM PO
WHERE Tenant = 17 -- Hospital X
AND Type = 2 -- Account
AND c0 = 'Gump'
AND c1 = 'State'
```

4 Evaluation

Because we implemented the approaches directly on the Java API of the benchmark *MTCB* [1], running the benchmark to evaluate them required no extra effort. For each approach we ran the main script ten times for each profile. The result is show in Table 2. For each metric we report the average (μ) and the coefficient of variation ($\frac{\sigma}{\mu}$), unless for *Size of disk*, because this is reported by the setup script, which was only run once for each approach for each profile. We also include the results for the baseline system provided by *MTCB*, a schema based implementation. We did not run this baseline for the *Medium* profile because we estimated that it would take over 48 hours for it to even finish running the

	Field Based	Field Based Alt	Row Based	Row Based Alt	Schema Based
	μ ($\frac{\sigma}{\mu}$)	μ ($\frac{\sigma}{\mu}$)	μ ($\frac{\sigma}{\mu}$)	μ ($\frac{\sigma}{\mu}$)	μ ($\frac{\sigma}{\mu}$)
TINY					
Size on disk (MB)	36	36	38	38	138
TDI created per minute	5,657 (0.08)	5,523 (0.08)	4,129 (0.21)	6,003 (0.14)	1,504 (0.47)
TDI loaded by ID per minute	108,797 (0.28)	136,086 (0.36)	49,685 (0.31)	51,279 (0.23)	144,585 (0.29)
Conjunctive searches per minute	641 (0.11)	515 (0.10)	63 (0.11)	22,000 (0.14)	85,461 (0.02)
Disjunctive searches per minute	263,994 (0.04)	60 (0.14)	63,924 (0.03)	11,665 (0.05)	665,173 (0.04)
SMALL					
Size on disk (MB)	299	298	328	328	5,471
TDI created per minute	4,163 (0.13)	4,893 (0.06)	4,365 (0.10)	8,702 (0.09)	2,442 (0.17)
TDI loaded by ID per minute	147,391 (0.31)	153,528 (0.24)	50,601 (0.41)	61,377 (0.37)	168,722 (0.21)
Conjunctive searches per minute	10 (0.00)	48 (0.28)	10 (0.03)	2,533 (0.01)	9,074 (0.06)
Disjunctive searches per minute	201,681 (0.30)	13 (0.04)	56,909 (0.21)	1,238 (0.01)	580,297 (0.06)
MEDIUM					
Size on disk (MB)	2,908	2,906	3,207	3,205	N/A
TDI created per minute	666 (0.03)	768 (0.04)	3,067 (0.54)	9,323 (0.02)	N/A
TDI loaded by ID per minute	132,308 (0.34)	142,661 (0.40)	54,808 (0.28)	62,950 (0.32)	N/A
Conjunctive searches per minute	105 (0.01)	4 (0.05)	32 (0.36)	260 (0.02)	N/A
Disjunctive searches per minute	200,981 (0.02)	4 (0.00)	60,136 (0.03)	124 (0.02)	N/A

Table 2. Main benchmark results for 10 runs for all profiles, showing the avarage (μ) and the coefficient of variation ($\frac{\sigma}{\mu}$).

setup script. The hardware that the benchmark was run on is a Centos 7 server with 32 GB RAM and an Intel Xeon E3-2200 Quad Core CPU.

The metrics *Aulbach compliance*, *Tenants created*, *Types created* and *Attributes created* were omitted from the results table because all of our approaches score 100% on these for all profiles. However, the Schema Based approach does fail the *Tenants created* and *Attributes created* metrics, as discussed in previous work [1].

All implementations pass the basic *Aulbach* compliance test, which means that they are capable of representing the running example by Aulbach et al. as discussed in Section 2.2.

In terms of *Size on disk*, the Row Based methods have about a 10% overhead in comparison with the Field Based methods for maintaining the PO Table. There is hardly any compensation in terms of the size of the Lookup Table, because nearly all attributes in *MTCB* are indexed. Both approaches improve greatly on the Schema Based approach. The reason that Schema Based is about 20 times as big for the *Medium* profile is that each schema adds its own duplicate metadata to the PostgreSQL data dictionary.

All our approaches are fully compliant with the metadata creation metrics *Tenants created*, *Types created* and *Attributes created*. This is because in our approaches this comes down to little more than adding a row to a table. In Schema Based expensive DDL operations have to be executed, which is why this approach fails to be compliant. It is also the reason that the setup script for Schema Based is so slow.

Massive differences can be seen in the results for the metrics *Conjunctive searches per minute* and *Disjunctive searches per minute*. The Schema Based Approach vastly outperforms the other approaches on these metrics. From the other approaches, the best achiever for the disjunctive search is Field Based, at 200,000 for the Medium profile. The worst achiever is Field Based Alt, at only 4 per minute. It is peculiar that this difference is so massive, because the Field Based Alternative uses a query that is functionally equivalent to the one used by Field Based. The reason is that in the Field Based Approach the PostgreSQL query planner fails to use the Number Value Index (Attribute, NumberValue, Row). Other relational databases may not make this mistake.

For the Conjunctive search, nearly all implementations perform quite poor. But aside from the Schema Based implementation, the Row Based Alternative clearly outperforms the others. The Row Based Alternative does not use the Lookup Table, but queries the unindexed PO Table directly. It turns out that this is the best strategy for the Conjunctive search in this benchmark. The reason for this is that the attribute values have very low selectivity: there are only $\sqrt[5]{n}$ distinct values for a search type that has n instances. In these cases, it is much faster for the DBMS to simply retrieve all instances for that type and tenant and iterate over all of them.

The *TDI created per minute* results shows an anomaly. The Row Based Alternative unexpectedly outperforms Row Based, but they only differ in how the conjunctive and disjunctive search were implemented. The reason is that the

TDI creation metric uses a disjunctive search to retrieve Master Data Type Instances by name. Because there are only 2 instances per tenant, this predicate also has a very low selectivity, so it is better to skip the index.

Some metrics show a quite high coefficient of variation. For example, the metric *TDI loaded per minute* shows values between 0.23 and 0.40. There are a lot of factors influencing the performance of PostgreSQL. This is one of the downsides of implementing the MTC-DB on an existing RDBMS. When building and benchmarking a system that was built from scratch, it should be much easier to get consistent test results. However, the variation is still low enough to draw conclusions from the results.

5 Conclusion

These benchmark results show that in general our approaches are viable. On nearly all metrics there is at least one approach that scores very well, except for the conjunctive search. However, the conjunctive search may be useful to benchmark, but is a very extreme scenario: in the Medium profile it is making a selection out of 1,000,000 records based on five attributes with extreme bad selectivity. In a production scenario, we could block such queries for regular users and only allow them for premium users.

It seems that each approach has its merits, so it is not possible to conclude that one approach is the best. For searching, sometimes it is best to use just the Lookup Table, and sometimes it is best to skip it completely and query the unindexed PO Table. We recommend future work to look into a hybrid approach that combines the Field Based and Row Based approaches. This hybrid approach should contain a smart query planner that uses statistics and machine learning to decide the best query to use.

References

1. W. van der Zijden, "Multi-tenant customizable database benchmark," 2017. Master's paper, University of Twente.
2. R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, 2014.
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 '08. Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1195–1206, ACM, 2008.
4. D. Jacobs and S. Aulbach, "Ruminations on multi-tenant databases," in *Datenbanksysteme in Business, Technologie und Web (BTW 2007), 12. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"*, DBIS, 2007.
5. D. Maier and J. D. Ullman, "Maximal objects and the semantics of universal relation databases," *ACM Transactions on Database Systems*, vol. 8, no. 1, pp. 1–14, 1983.
6. S. Chickerur, A. Goudar, and A. Kinnerkar, "Comparison of relational database with document-oriented database (mongodb) for big data applications," in *2015*

8th International Conference on Advanced Software Engineering & Its Applications (ASEA), pp. 41–47, IEEE, 2015.

7. C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, “A comparison of a graph database and a relational database,” in *ACM SE '10 Proceedings of the 48th Annual Southeast Regional Conference*, ACM, 2010.
8. L. Wevers, “A persistent functional language for concurrent transaction processing,” Master's thesis, University of Twente, 2012.
9. S. Manegold, M. L. Kersten, and P. Boncz, “Database architecture evolution: Mammals flourished long before dinosaurs became extinct,” *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1648–1653, 2009.
10. C.-P. Bezemer and A. Zaidman, “Challenges of reengineering into multi-tenant saas applications,” *Delft University of Technology Software Engineering Research Group. Technical Report Series*, 2010.