Data Storage in Clojure

Options, Analysis, and Assessment

Brian Abbott, February 20th, 2019

Note: this document is available online at:
https://docs.google.com/document/d/11zGRKTfzAK4GT8Kv41NyQebY89zAjk4Xs9uBzc_iAQY/edit?usp=sharing

Revision History

Version	Notes	Date
0.1	A Strawman Alpha	02.15.2019

Overview

Falling out of the developer standup meeting on 02.15.2019 regarding our choices and, potential migration away from and too alternative persistent data storage engines now, or perhaps at various points in the future, it is valuable for us to do an extended exploratative evaluation of what a variety of data-storage options might look like for us as well as, to what degree of compatibility might we find between data-storage platforms and perhaps additionally, what frameworks, toolkits and supporting development and production applications and tool-sets might we find available for any one potential or currently selected data storage platforms.

It is also good to keep in mind, while currently we are in a position that is best and ideal for us to have only one single primary data-storage platform, considering common application-platform growth paths as well as taking into consideration the nature of Health-Care Information Systems and Processing, it is easy for one to arrive at the consideration that we are more then likely to find ourselves in a future position of needing to integrate with multiple data-platforms simultaneously and, at equal levels of integration and application supporting functionality.

Example Application and Data Model

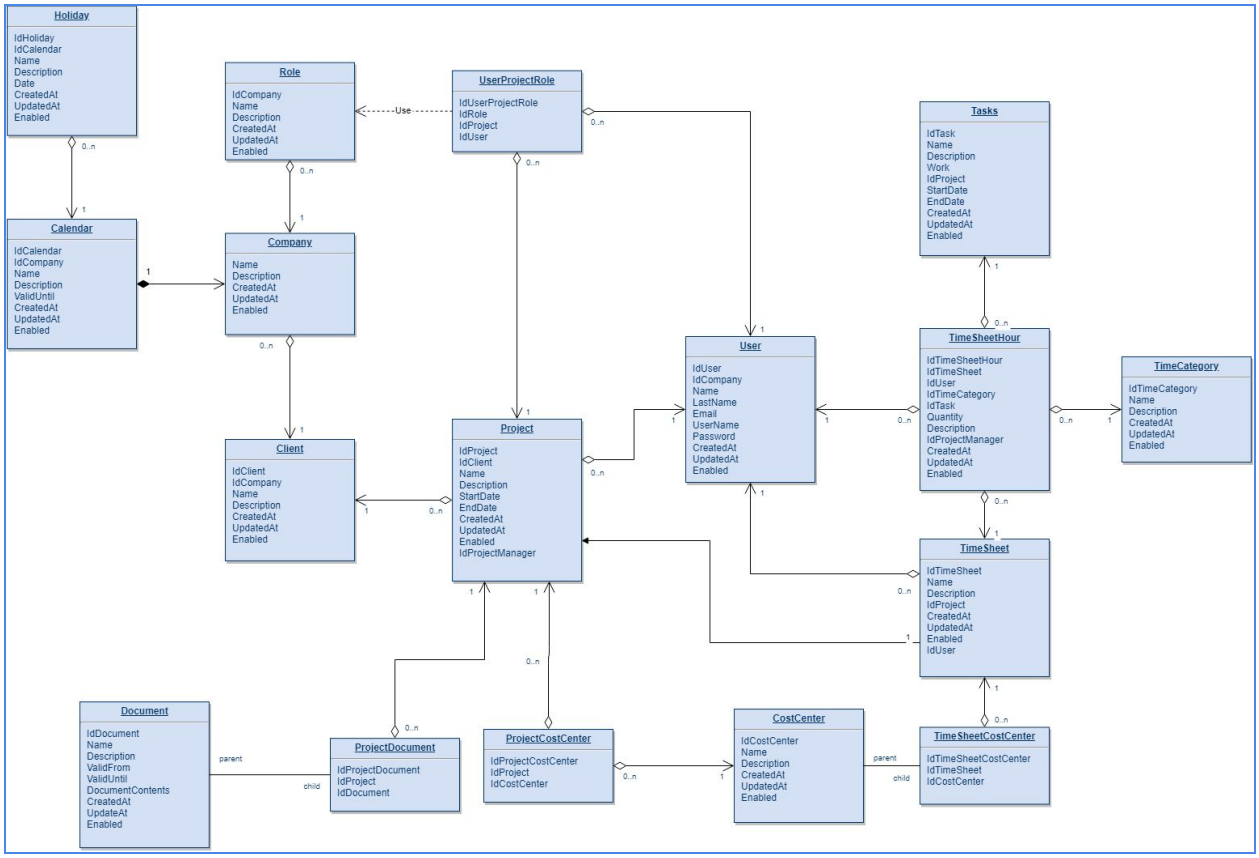
In order to find a basis for comparative analysis, it is ideal for us to synthesize a common application data model that we then re-render for each data-platform to which we seek to understand and then, view that data platform through the lense of the degree of complexity in rendering the application on each platform. To that end, we will capture here a simple User-Registration Application Data-Model that we will use throughout our data-platform analysis.

Requirements in Showcase Application and Data-model

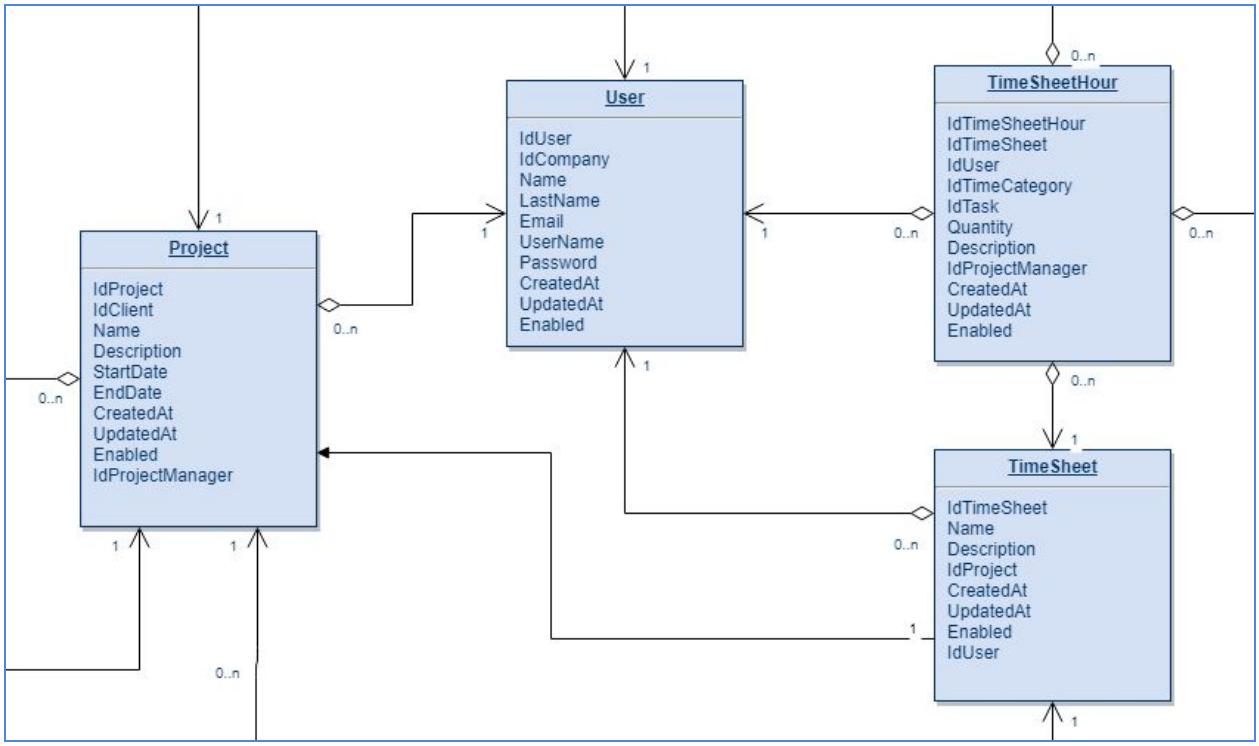
TODO: Fill me out...

Showcase Data-model

In thinking about and trying to identify a nice example application model for our research, I came across this already developed and delivered model which was part of the template diagram from Draw.io for the Database Model Diagram. This template can be seen below and, is available in its raw web format from draw.io (https://www.draw.io/) by creating a new diagram and selecting the Software Template Collection and clicking on software/database_1.xml.



This is a data-model for project-management including cost-accounting. Now, while this is quite nice - albeit somewhat hard to see in this document - it is a bit more code then we want to type in and recreate for having a small experimental data-application. Therefore, we will focus on a simplified subset of the data-model for our purposes.



This is better. It's a subset that's highly referential, integrated, and interdependent. In simpler terms - it has the shape (I think) that we're looking for. We can write some complex data-model requirements and corresponding queries and actions against this for the four basic CRUD operations and beyond. Let's go with it. We will add in the primitive field-types, et al later as we get going with each data-store, as the types might change there anyway.

Our Data Story

Now that we've outlined our schema, we need to come up with some data to put in it or rather, a story that will drive the data we put into it. The project management and cost-accounting application under test is for the BoingBoing^2 Aircraft Corporation, located in SeaTown USA. They design and build super secret aircraft using advanced space-jet technologies. They have to keep their teams small to manage the classified nature of their work. To that end, they employ 2 (project) managers, 4 workers, and have 16 active projects and to date have issued 256 billable timesheets for a total of 65536 time-sheet hours.

Our Project Managers are Sally and Susan, who manage four workers, Bob, Larry, Moe, Tyler who currently work on 16 projects:

- A-12 Avenger II
- AA160 Hummingbird
- Bird of Prey

- Condor
- F/A-XX Program
- Pelican
- Phantom Ray
- Quad TiltRotor (QTR)
- X-32
- X-37
- X-40
- X-45
- X-48
- X-53 Active Aeroelastic Wing
- YF-23
- ZZX-51 Waverider

Projects are split evenly between Sally and Susan in Alphabetical order except for the Avenger and Waverider projects, which they both manage together. Given all that, we have the following table that outlines our data-instance structure.

Project	Managed By	Worker(s)	Timesheets	Hours
A-12 Avenger II	Sally, Susan	Bob, Larry, Moe, Tyler	32	
AA160 Hummingbird	Susan	Bob, Larry	14	
Bird of Prey	Sally	Larry, Moe	14	
Condor	Susan	Moe, Tyler	14	
F/A-XX Program	Sally	Tyler, Bob	14	
Pelican	Susan	Bob, Moe	14	
Phantom Ray	Sally	Larry, Bob	14	
Quad TiltRotor (QTR)	Susan	Moe, Larry	14	
X-32	Sally	Tyler, Moe	14	
X-37	Susan	Bob, Tyler	14	
X-40	Sally	Larry, Tyler	14	
X-45	Susan	Moe, Bob	14	

X-48	Sally	Tyler, Larry	14	
X-53 Active Aeroelastic Wing	Susan	Bob, Larry	14	
YF-23	Sally	Larry, Moe	14	
ZZX-51 Waverider	Sally, Susan	Bob, Larry, Moe, Tyler	32 Timesheets	

Overview of our Analysis Process

For each data storage platform, we will run, albeit somewhat mechanically, through a same series of operations as we investigate each target platform. This will produce a result that will read somewhat monotonous but, provide a strong basis for an equality of compare. The demonstrative tasks that will drive our analytical operations are as follows:

- Native Data Operations
- Data Operations with Clojure Core APIs
- Data Operations via Frameworks and Libraries

First, by data-operations, we mean Data-Model or Schema representation and augmentation operations, such as create, and modify as well as to be able to insert, update, and delete records within our schema - the common operations performed against a database. Where a given data-platform does not allow or, provides a significantly different mechanism for a common data-operation, we will highlight it and, provide an analysis of the reason for the difference as well as to describe and illustrate the alternative operations native to the given platform.

Native Data Operations

By native, we mean to utilize and demonstrate operations in the raw language syntax of each data-platform. Additionally, in these exercises, we desire to have an affinity for methods that are the most germain or resident to the target platform over those that are generic or common. While we may or may not want to interact with our data-store platforms this way in production, we are motivated in these analytical exercises to come to an understanding of the platform that minimizes the distance from the platforms most intrinsic and raw ways, methods, tastes and preferences for doing things while maximizing our distance from any one of the many given standards for data-storage, data-querying, and data-retrieval of which, there are many (i.e. SQL, RDF/SPARQL, XML) - it is not our goal to know common intersections but, to establish an

intimate familiarity with each platform. An example of this might be, while Postgres might support standard SQL, we seek to understand the internal operations of the engine, as well as the enhancements and nuances made available to us by the Postgres native PG-SQL Query Language.

Data Operations with Clojure Core APIs

The data operations captured and discussed within these sections will be either with the clojure java.jdbc API or, with the native libraries that each corresponding platform either provides, points-to, or sanctions as, “the clojure way of connecting to and operating upon this data engine”. For example, while MongoDB does not itself provide a Clojure API, Monger from November-Rain (http://clojuremongodb.info/) is the community accepted library to connect to and operate on a MongoDB instance from within Clojure.

Data Operations via Frameworks and Libraries

This list and, this aspect of our endeavour is meant to be somewhat exhaustive. The reasoning and motivation for this is that, these components, if adopted, will become the intermediary between our application code and our datastore. Depending on the provided model of interaction, the code utilizing and dependant upon these libraries could become quite ubiquitous and pervasive throughout our codebase and, therefore, it could became both a costly as well as a high-risk if not impossible endeavour to transition from one library to another, perhaps even more difficult and concerning then it might be to move from one data storage platform to another. Therefore, we want to ensure that the SQL/Data Querying framework(s) that we select, provide a maximum adaptability between a variety of Data Storage Engines.

The instance I reflect on when coming to this conclusion is the LINQ technology incorporated within the .NET Framework. LINQ makes it exceedingly easy to directly code, from anywhere within your application code, at anytime, against your database, as if you are simply calling functions against object instances. This generally means that, once you adopt LINQ and the Entity Framework within your .NET project, data-access-layer client code, directly hitting the primary storage engine, quickly ends up all over your application and is virtually impossible to remove in larger production codebases without either taking an inordinate amount of time or, inducing a choking degree of risk within your release to remove. Therefore, I feel that it is worth the while and the effort to fully explore and exhaust our options in this regard, as many of these frameworks appear to be quite powerful and capable and, therefore, deserve a degree of analysis and understanding that is on par with their capability. -*”With great power, comes great responsibility”* and, therefore it is on us to know the extent and limitations of these frameworks and then, with that knowledge to choose one or orchestrate multiple and unleash that power unto and within the heart of our core frameworks. To that end, the list of frameworks that we shall seek to understand and capture their interaction with each of our data-storage engines is as follows.

- HugSQL
- Korma
- Honey SQL
- SQLingvo
- oj
- Suricatta
- aggregate
- Toucan

The following frameworks were considered but dropped upon further investigation. The reasoning behind their exclusion will be described within their respectives sections.

- Yesql

NOTE: TODO: Rather then including this analysis action within each target, we might include it outside individual data-platforms at the end in a consolidated compare and contrast, if it becomes too repetitive or verbose for the desired density and length of the document (something to be read and understood in a single 30 minute reading). This decision will be made as we go about understanding and illustrating the first couple data-store engines.

HugSQL

The name Hug means to embrace SQL, effectively giving it a Hug. This is befitting as it captures the central philosophy behind Hug. Hug also seems to be widely referenced and adopted however, it also appears to be the least interesting in that it is primarily mappings from SQL files to Clojure Functions through a three step coding process. First, we define a SQL file with mapping definitions captured as SQL comments.

```
-- :name create-characters-table
-- :command :execute
-- :result :raw
-- :doc Create characters table
--  auto_increment and current_timestamp are
--  H2 Database specific (adjust to your DB)
create table characters (
  id         integer auto_increment primary key,
  name       varchar(40),
  specialty  varchar(40),
  created_at timestamp not null default current_timestamp
```

```
)
```

Then we call Hug to read our data definitions, causing functions to be created for us that are a product derived from the mapping definitions supplied in the SQL files.

```
(ns db.characters
  (:require [hugsql.core :as hugsql]))

(hugsql/def-db-fns "db/sql/characters.sql")
(hugsql/def-sqlvec-fns "db/sql/characters.sql")
```

And finally, we call database functions from within our application code that produce SQL statements available for execution through and active database connection.

```
(characters/characters-by-ids-specify-cols-sqlvec
  {:ids [1 2], :cols ["name" "specialty"]})

;;=> ["select name, specialty from characters
     where id in (?,?)",1,2]
```

One benefit of Hug is its huge and well maintained library of developer documentation, available at: https://www.hugsql.org/. The codebase for HugSQL can be found at: https://github.com/layerware/hugsql.

Korma

Korma is perhaps the oldest, and certainly most popular SQL DSL for Clojure. It uses a macro-based DSL to represent queries as if they were Clojure forms:

```
(defentity users)

(select users
  (where {:active true})
  (order :created)
  (limit 5)
  (offset 3))
```

Korma is designed to replace java.jdbc for all data-store operations. It provides a comprehensive API for declaring database relationships and properties, and you'll probably

never need to drop into SQL unless you want to use some database-specific features. That said, you can still use the sql-only macro to retrieve a SQL vector from Korma to do with what you will. Another positive attribute of Korma is its extensive documentation available at http://sqlkorma.com/. The codebase for Korma can be found at: https://github.com/korma/Korma

Honey SQL

HoneySQL represents queries as a map, which you can then format into an SQL string for use with data.jdbc:

```
(h/format {:select [:*]
           :from [:users]
           :where [:= :active true]
           :order-by [:created]
           :limit 5
           :offset 3})
```

It also provides helper functions for building said map. Each function works like an assoc, so you can pipe them together, which ends up looking very korma-y

```
(-> (select :*)
    (from :users)
    (where := :active true)
    (order-by :created)
    (limit 5)
    (offset 3)
    (h/format))
```

Both of those produce a vector containing an SQL string and any bound parameters, which you can plug (in this case) directly into java.jdbc's query function to get a result back. HoneySQL can be found here: https://github.com/jkk/honeysql.

SQLingvo

SQLLingvo works a lot like HoneySQL, but defines a slightly different-looking DSL and eschews the intermediary maps (actually, it does use maps if you try executing the functions alone, but they're a lot less readable than HoneySQL's). Instead, the base query functions (select etc.) can contain forms modifying the query, and it applies the transformations internally rather than externally as in honey.

```
(select db [*]
        (from :users)
        (where '(= :active true))
        (order-by :created)
        (limit 5)
        (offset 3))
```

In the code above, the `db` argument is defined by a call to `sqlingvo.db.postgresql`, which you might expect means a query is executed - This is not the case! The db argument is there to tell sqlingvo how to quote the query, which could be a nice feature if you use a particularly uptight database. The function above produces a SQL vector for use with query just as in the honeysql examples. SQLLingvo can be found here: https://github.com/r0man/sqlingvo

oj

oj (all lowercase) is similar to HoneySQL, representing queries as Clojure maps. The biggest difference is that oj provides a facility (exec) for executing the maps directly, rather than rendering them as strings and passing them to jdbc:

```
(oj/exec {:table :users
          :where {:active true}
          :order-by :created
          :limit 5
          :offset 3} db-spec)
```

Like honeysql, oj also provides utilities for not using a map. However, they're so basic that reading their implementation feels almost sarcastic. I tried to render the above example, but offset is not present (and if it was, it would just be #(assoc %1 :offset %2)). OJ is primarily about using maps to express SQL operations. At the outset, this feels very much like the HTML libraries provided by Luminus. oj can be found here: https://github.com/taylorlapeyre/oj

Suricatta

Suricatta is a combination jdbc helper (providing some nice refinements over jdbc's api) and Korma-esque query DSL. Here's what the latter looks like:

```
(-> (d/select :email)
    (d/from :users)
    (d/where ["active = ?" true])
    (d/order-by :created)
```

```
    (d/limit 5)
    (d/offset 3)
    (sqlvec))
; => ["select email from users where (active = ?) order by created asc
limit ? offset ?" true 5 3]
```

Suricatta can be found here: https://github.com/funcool/suricatta

Aggregate

Aggregate is interesting in that it builds from concepts specified in the widely acclaimed book "Domain Driven Design" by Eric Evans. Aggregate provides three core database operation functions and then takes sets of definitions in order to execute the functions. From Martin Fowler on the DDD Aggregate Pattern: *"Aggregate is a pattern in Domain-Driven Design. A DDD aggregate is a cluster of domain objects that can be treated as a single unit. An example may be an order and its line-items, these will be separate objects, but it's useful to treat the order (together with its line items) as a single aggregate."*

```
(ns agg-app
  (:require [aggregate.core :as agg]))


(agg/load er-config db-spec entity-keyword id)
(agg/save! er-config db-spec data)
(agg/delete! er-config db-spec data)
```

Saving a project-definition in aggregate looks like this.

```
(def project (agg/save! er @h2/db-con :project
              {:name "Learning Clojure"
               :customer {:name "Big Company"}
               :tasks [{:desc "Buy a good book" :effort 1}
                       {:desc "Install Java" :effort 2}
                       {:desc "Configure Emacs" :effort 4}]
               :members [{:name "Daisy"}
                         {:name "Mini"}]
               :manager {:name "Daisy"}}))
```

aggregate can be found here: https://github.com/friemen/aggregate.

Toucan

Toucan builds on HoneySQL to incorporate design ideas from Korma while providing an ORM-esque experience in Clojure. Toucan provides the better parts of an ORM for Clojure, such as simple DB queries, flexible custom behavior when inserting or retrieving objects, and easy hydration of related objects. Toucan builds on top of clojure.java.jdbc and HoneySQL. The primary motivation behind Toucan was to bring some of the missed conveniences of Korma over to HoneySQL. The Toucans ORM-like behavior can be demonstrated in this sample.

```
;; define the User model
(defmodel User :user
  IModel
  (types [this]
    {:status :keyword}))

;; Insert a new User
(db/insert! User :name "Cam", :status :new)

;; Fetch User 200
(User 200)
```

The only requirement to making code like this work is to first define a JDBC connection to the data store engine.

```
(require '[toucan.db :as db])

(db/set-default-db-connection!
  {:classname   "org.postgresql.Driver"
   :subprotocol "postgresql"
   :subname     "//localhost:5432/my_db"
   :user        "cam"})
```

Toucan can be found here: https://github.com/metabase/toucan

Considered but dropped

Yesql

Yesql was considered for evaluation as it was quite frequently mentioned on the web however, upon investigation, I found that the github landing page had the following text in the readme and, therefore I am taking it out as a contender for our purposes.

“Status: Frozen. Maintainer sought.
Tested with Clojure 1.5-1.9-alpha20, but there will be no new development unless a maintainer steps up.
...
(You might also consider hugsql which is philosophically similar and actively maintained.)”

Overview of our Data-Store Analysis Targets

Everything up to this point has setup for us a general framework for data-platform analysis, from which we can add or remove any given data-platform, now or at any point in the future. To some extent, that is a core goal of this document, to outline and prescribe for us a core framework that can be utilized at any point in time to rapidly understand how a considered data platform or data-framework API will interact and integrate with our platform and with the clojure programming language.

TODO: Include requirements/criteria for inclusion…

Criteria for inclusion:

- Popularity
- Compatibility with Clojure

At the time of this initial writing, we seek to understand through our analysis process, the following data-storage platforms:

- PostgreSQL
- MongoDB
- Datomic

PostgreSQL - a case for PostgreSQL

One of our top, key targets for understanding is the PostgreSQL Open Source Relational Engine. Postgres is a highly utilized, well know, well supported and highly capable relational engine.

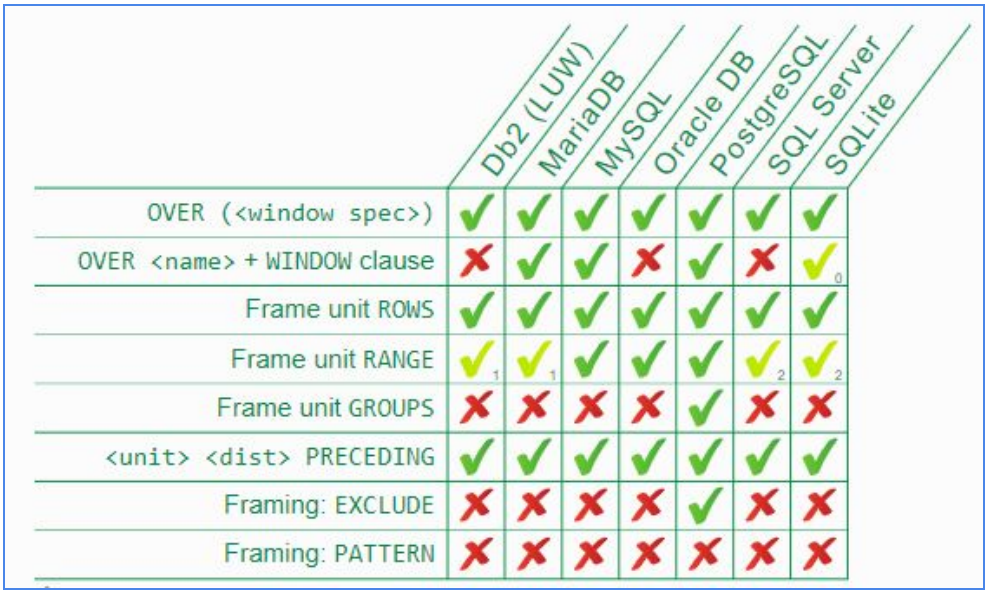
This chart, provided by the db-engines.com ranking engine, which utilizes a custom application specific search engine to identify database popularity, clearly shows Postgres holding strong as the number four most utilized database in the world.

Rank			DBMS	Database Model	Score		
Feb `19	Jan `19	Feb `18			Feb 19	Jan 19	Feb 18
1.	1.	1.	Oracle	Relational, Multi-model	1264	-4.82	-39.26
2.	2.	2.	MySQL	Relational, Multi-model	1167	+13	-85.18
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1040	-0.21	-81.98
4.	4.	4.	PostgreSQL	Relational, Multi-model	473	+7.45	+85.18
5.	5.	5.	MongoDB	Document	395	+7.91	+58.67
6.	6.	6.	IBM Db2	Relational, Multi-model	179	-0.43	-10.55
7.	7.	↑ 8.	Redis	Key-value, Multi-model	149	+0.43	+22.43
8.	8.	↑ 9.	Elasticsearch	Search, Multi-model	145	+1.81	+19.93
9.	9.	↓ 7.	Microsoft Access	Relational	144	+2.41	+13.95
10.	10.	↑ 11.	SQLite	Relational	126.17	-0.63	+8.8

In addition to being quite popular as a data storage platform, Postgres is one of the more feature rich, capable databases. While we discussed earlier the desire to capture resident or intrinsic behaviors over industry standard ones, one strong highlight of Postgres is its full and early on support for new SQL standards. We will spend some time here highlighting some of the functionality-comparison charts that highlight the breadth and depth of Postgres supported standard SQL functionality over its competitors.

SQL:2011 Window Function row-visibility via the Over Clause

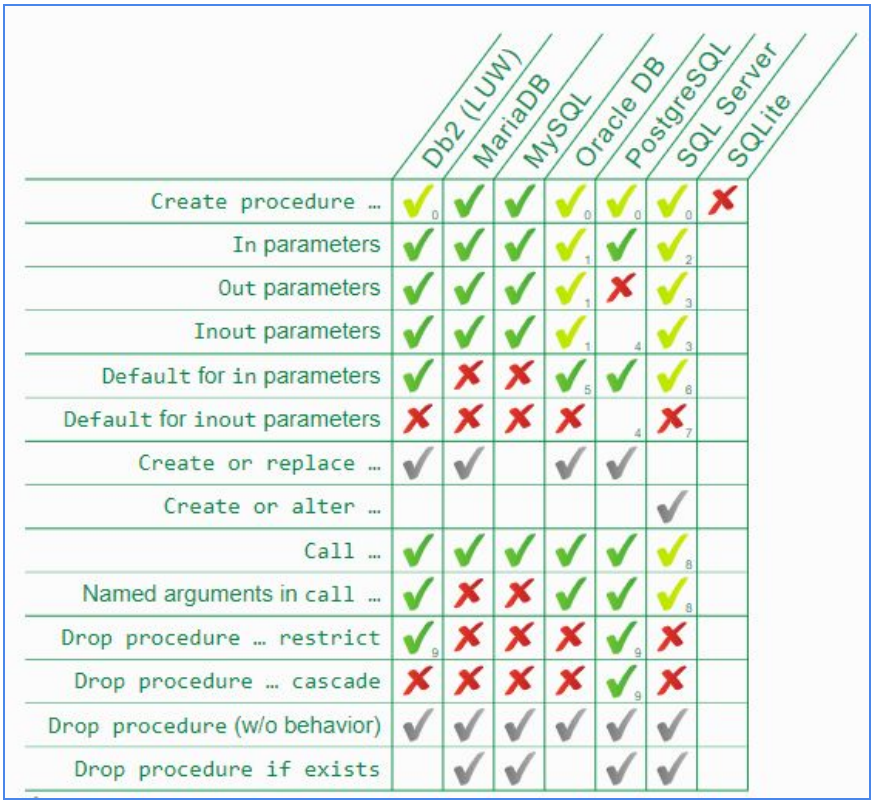
Window Functions were originally standardized in SQL:2003 and n the SQL:2011 Standard, support was introduced for what is referred to as the Over Clause. An over clause selects or sets which rows or, distances from a given row are available to a Window Function. As we can see from the support matrix below, PostgreSQL version 11, the most recent version, by far exceeds support beyond any other traditional relational database engine.



	Db2 (LUW)	MariaDB	MySQL	Oracle DB	PostgreSQL	SQL Server	SQLite
OVER (<window spec>)	✓	✓	✓	✓	✓	✓	✓
OVER <name> + WINDOW clause	✗	✓	✓	✗	✓	✗	✓[0]
Frame unit ROWS	✓	✓	✓	✓	✓	✓	✓
Frame unit RANGE	✓[1]	✓[1]	✓	✓	✓	✓[2]	✓[2]
Frame unit GROUPS	✗	✗	✗	✗	✓	✗	✗
<unit> <dist> PRECEDING	✓	✓	✓	✓	✓	✓	✓
Framing: EXCLUDE	✗	✗	✗	✗	✓	✗	✗
Framing: PATTERN	✗	✗	✗	✗	✗	✗	✗

Create Procedure Support

Create Procedure support was only added in Postgres 11 (although User Functions - what we normally think of for PG-SQL have been there forever) however, while being a late comer to the game, Postgres, aside from the corporate backed DB2, currently boasts the greatest degree of support for the Create Procedure and associated syntactical artifacts.

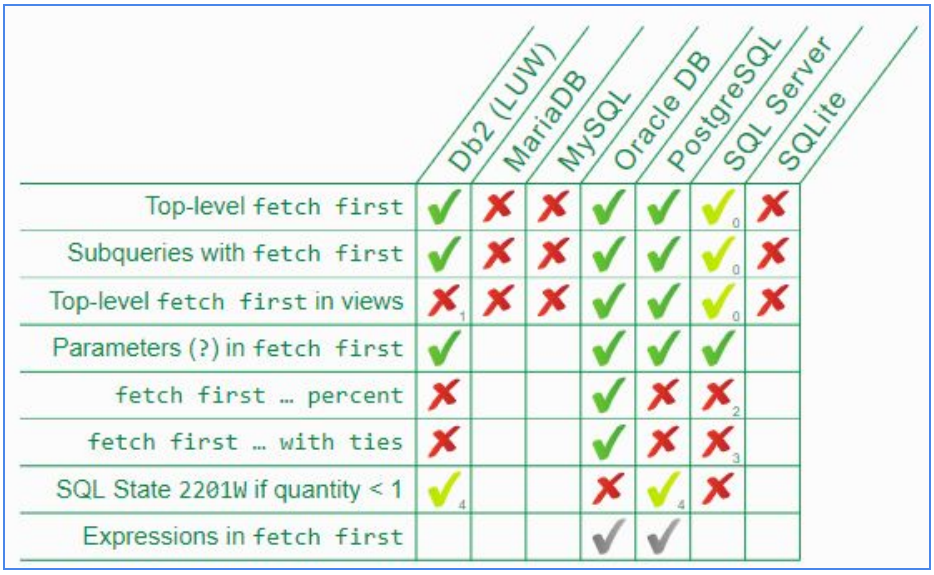


	Db2 (LUW)	MariaDB	MySQL	Oracle DB	PostgreSQL	SQL Server	SQLite
Create procedure …	✓[0]	✓	✓	✓[0]	✓[0]	✓[0]	✗
In parameters	✓	✓	✓	✓[1]	✓	✓[2]	
Out parameters	✓	✓	✓	✓[1]	✗	✓[3]	
Inout parameters	✓	✓	✓	✓[1]	[4]	✓[3]	
Default for in parameters	✓	✗	✗	✓[5]	✓	✓[6]	
Default for inout parameters	✗	✗	✗	✗	[4]	✗[7]	
Create or replace …	✓	✓		✓	✓		
Create or alter …						✓	
Call …	✓	✓	✓	✓	✓	✓[8]	
Named arguments in call …	✓	✗	✗	✓	✓	✓[8]	
Drop procedure … restrict	✓[9]	✗	✗	✗	✓[9]	✗	
Drop procedure … cascade	✗	✗	✗	✗	✓[9]	✗	
Drop procedure (w/o behavior)	✓	✓	✓	✓	✓	✓	
Drop procedure if exists		✓	✓		✓	✓	

Note that, the empty box marked with a number 4 in subscript, on the line titled “Inout parameters”, was unavailable to be tested as the JDBC driver utilized currently does not support the functionality.

Fetch-First Support

Again, like the two previous feature-matrices, the only databases to outperform Postgres are expensive proprietary systems with massive corporate backing. In this chart, we can see that, other then Oracle, Postgres offers the most extensive and complete support of the SQL-Standard Fetch First N syntax, the non-proprietary variance to the limit clause.



	Db2 (LUW)	MariaDB	MySQL	Oracle DB	PostgreSQL	SQL Server	SQLite
Top-level `fetch first`	✓	✗	✗	✓	✓	✓[0]	✗
Subqueries with `fetch first`	✓	✗	✗	✓	✓	✓[0]	✗
Top-level `fetch first` in views	✗[1]	✗	✗	✓	✓	✓[0]	✗
Parameters (`?`) in `fetch first`	✓			✓	✓	✓	
`fetch first … percent`	✗			✓	✗	✗[2]	
`fetch first … with ties`	✗			✓	✗	✗[3]	
SQL State `2201W` if quantity < 1	✓[4]			✗	✓[4]	✗	
Expressions in `fetch first`				✓	✓		

We will go into more details regarding Postgres in the dedicated section however, these details offer just a glimpse as to the strength and viability offered by Postgres. In the realm of traditional Relational, RDBMS platforms, Postgres is quite clearly a leading contender.

MongoDB

A case for MongoDB - NOTE: We may not include this platform in our studies.

Datomic

TODO: This section needs to be cleaned up and is not finished - I am trying to find better, more appropriate use, presentation and comparative resources that I can utilize to build an objective (i.e. first-eyes) case for Datomic (by first eyes, I mean - to present these cases for each platform as if the reader has never seen or heard of it, at least relative to the other platforms - these sections are meant to read like short brochures).

A case for Datomic
This database platform, more than any other, has the total and absolute backing of the Clojure Community - this is the clojure database.

Now, being the

Aside from being THE Clojure Database, created and sanctioned by the creator of Clojure, Datomic genuinely does have a very unique and novel data-model. Because of this core model, Datomic enables the handling of row-oriented, column-oriented, graph, and hierarchical data within a single system. Additionally, Datomic is natively or intrinsically time-aware.

Datomic has three product variants, which means, for the purposes of our analysis, we have to somewhat pick one and run with it. For this, we will select Datomic Cloud. The reasoning is that, first, we have already started a transition to Datomic Cloud, it very much makes the most sense.

As stated by Cognitect, the features of Datomic Cloud are as follows:

- Indelibility and Auditability
- Flexible Data Model
- ACID Transactions
- Powerful Query Language
- Horizontal Read Scaling

In addition to those core engine capabilities, Datomic's close integration with AWS enables offloading of administrative tasks and, perhaps most importantly of operating and scaling burdens, removing common operational chores such as hardware provisioning, setup and configuration, as well as capacity planning. While it is true that other database platforms will run on AWS within EC2 containers and, Amazon has built their own data-platform offerings through the RDS organization, is is nice to see a third party data-platform work to integrate as deeply with AWS as Amazon's own data platforms do.

What sorts of applications is Datomic designed for?

Datomic is designed as a general-purpose fully ACID transactional data of record system. It is a good fit for systems that store valuable information of record, require developer and operational flexibility, need history and audit capabilities, and require read scalability. Some examples of successful Datomic use cases include transactional data, business records, medical records, financial records, scientific records, inventory, configuration, web applications, departmental databases, and cloud applications.

PostgreSQL

PostgreSQL Overview

The PostgreSQL Datamodel

The PostgreSQL Native Query Language

PostgreSQL Toolsets, Applications, and Production Support, etc.

Native Data Operations

Schema Representation Operations

The following set of pg-sql commands create our schema within our PostgreSQL instance.

Our first operation is to create what we will call the database or schema, which we will call for the Postgres exercise, “nomihealth-datastore-pgsql”. We can create the database with the following PG-SQL code.

```
-- Database: nomihealth-datastore-ana

-- DROP DATABASE "nomihealth-datastore-pgsql";

CREATE DATABASE "nomihealth-datastore-pgsql"
    WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    LC_COLLATE = 'English_United States.1252'
    LC_CTYPE = 'English_United States.1252'
```

```
    TABLESPACE = pg_default
    CONNECTION LIMIT = -1;
```

Now that we have our database, we want to begin creating our tables. The first table that we will create is the Project Table. The project table is defined by the following PG-SQL code.

```
CREATE TABLE public."Project"
(
    "IdProject" integer NOT NULL,
    "Name" text NOT NULL,
    "Description" text,
    "StartDate" date,
    "EndDate" date,
    "CreatedAt" abstime,
    "UpdatedAt" abstime,
    "Enabled" boolean,
    "IdProjectManager" integer NOT NULL,
    PRIMARY KEY ("IdProject")
)
WITH (
    OIDS = FALSE
);

ALTER TABLE public."Project"
    OWNER to postgres;
```

Now we will create our User table, which, in reality, we should have done first since the Project table has a foreign key constraint on its IdProjectManager column to a User row. But, that is okay, we want to see what happens when we mess up and, how easy is it to resolve by adding only the fragment of our necessary update later on. We will make the same mistake with the other databases as we move through them. Here is the SQL required to create our User table.

```
CREATE TABLE public."User"
(
    "IdUser" integer NOT NULL,
    "IdCompany" integer,
    "Name" text NOT NULL,
    "LastName" text NOT NULL,
```

```
    "Email" text NOT NULL,
    "UserName" text NOT NULL,
    "Password" text NOT NULL,
    "CreatedAt" abstime NOT NULL,
    "UpdatedAt" abstime NOT NULL,
    "Enabled" boolean NOT NULL,
    CONSTRAINT "user-id" PRIMARY KEY ("IdUser")
)
WITH (
    OIDS = FALSE
);

ALTER TABLE public."User"
    OWNER to postgres;
```

Create TimeSheet Table

```
CREATE TABLE public."TimeSheet"
(
    "IdTimeSheet" integer NOT NULL,
    "Name" text NOT NULL,
    "Description" text,
    "IdProject" integer NOT NULL,
    "CreatedAt" abstime,
    "UpdatedAt" abstime,
    "Enabled" boolean,
    "IdUser" integer,
    CONSTRAINT "timesheet-id" PRIMARY KEY ("IdTimeSheet"),
    CONSTRAINT "fk-project" FOREIGN KEY ("IdProject")
        REFERENCES public."Project" ("IdProject") MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID,
    CONSTRAINT "fk-user" FOREIGN KEY ("IdUser")
        REFERENCES public."User" ("IdUser") MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
```

```
    OIDS = FALSE
);

ALTER TABLE public."TimeSheet"
    OWNER to postgres;
```

Create TimeSheetHour Table

```
CREATE TABLE public."TimeSheetHour"
(
    "IdTimeSheetHour" integer NOT NULL,
    "IdTimeSheet" integer NOT NULL,
    "IdUser" integer NOT NULL,
    "IdTimeCategory" integer,
    "IdTask" integer,
    "Quantity" numeric,
    "Description" text,
    "IdProjectManager" integer NOT NULL,
    "CreatedAt" abstime,
    "UpdatedAt" abstime,
    "Enabled" boolean,
    CONSTRAINT "timeSheetHour-id" PRIMARY KEY ("IdTimeSheetHour"),
    CONSTRAINT "fk-timesheet" FOREIGN KEY ("IdTimeSheet")
        REFERENCES public."TimeSheet" ("IdTimeSheet") MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT "fk-user" FOREIGN KEY ("IdUser")
        REFERENCES public."User" ("IdUser") MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT "fk-projectManager" FOREIGN KEY ("IdProjectManager")
        REFERENCES public."User" ("IdUser") MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
);
```

```
ALTER TABLE public."TimeSheetHour"
    OWNER to postgres;
```

And, that does it! We have our database and, four tables. Now what would be nice would be to point a live ERD tool at our Postgres instance and verify that it looks like our originally specified diagram. To Be Continued for now.

Oh, whoops, we forgot to go back and add our constraint that we had forgotten in the first table operation. We need a Foreign Key Constraint from Project to User on the IdProjectManager field, let's do that now and see how hard it is to make the update!

```
ALTER TABLE public."Project"
    ADD CONSTRAINT "fk-projectManager" FOREIGN KEY ("IdProjectManager")
    REFERENCES public."User" ("IdUser") MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION;
CREATE INDEX "fki_fk-projectManager"
    ON public."Project"("IdProjectManager");
```

Not too bad! That updates (via an Alter Table statement) the Project table to capture a foreign-key constraint on the User table, in order to ensure that every projects Project-Manager field points to an active and real user.

We finally have our complete and total schema definition, expressed, as PG-SQL statements. Now we are ready to go and create some data via PG-SQL INSERT operations.

Native Insert Operations

First a Test Insert

First we will need to create our Users, as we learned in the previous exercise with our Schema, that table is the root of our hierarchy however before we begin inserting the users we that we outlined in our Data-Story section, lets first create a test user, which we will then delete, just to make sure we know what we are doing and, that our table structure and field definitions were created and set-up correctly.

```
INSERT INTO public."User"(
    "Name", "LastName", "Email", "UserName", "Password", "CreatedAt",
```

```
    "UpdatedAt", "Enabled")
VALUES ('test.name', 'test.lastName', 'test.email',
        'test.userName', 'test.password', now(), now(), true);
```

And … Wham! We get an error once we execute that script

```
postgres-# ERROR:  null value in column "IdUser" violates not-null
constraint DETAIL:  Failing row contains (null, null, test.name,
test.lastName, test.email, test.userName, test.password, 2019-02-19
05:09:12-08, 2019-02-19 05:09:12-08, t).
SQL state: 23502
```

From this, we can see that we did something wrong in the setup of our original schema. It appears that the Primary-Key constraint for the IdUser column is not being activated on INSERT operations and creating an auto-generated, unique, primary key value as we expected by the original table definition.

Lets resolve that, not only for User but, for all of our tables. To do this, we will need to drop our foreign key constraints, alter the Id<TableName> fields, and then re-apply the foreign key constraints.

As it turns out, the correct syntax for creating our Primary Key constraints is as follows:

```
CREATE TABLE TableName (
    Id<TableName> serial primary key,
     ...
)
```

Now that we know that, lets just drop our Foreign Key constraints and recreate our tables with the correct syntax for the Primary Key definitions. First, we'll drop our Foreign Keys.

```
ALTER TABLE public."Project" DROP CONSTRAINT "fk-user";

ALTER TABLE public."TimeSheet" DROP CONSTRAINT "fk-project";
ALTER TABLE public."TimeSheet" DROP CONSTRAINT "fk-user";

ALTER TABLE public."TimeSheetHour" DROP CONSTRAINT "fk-projectManager";
ALTER TABLE public."TimeSheetHour" DROP CONSTRAINT "fk-timesheet";
ALTER TABLE public."TimeSheetHour" DROP CONSTRAINT "fk-user";
```

Now, we just to to re-execute our table create scripts with the Primary Key's defined as constraints removed and, to modify the column definitions of the Id<Table> columns as we re-execute our create scripts. The Primary Key Constraint Definition appears as follows:

```
CONSTRAINT "timeSheetHour-id" PRIMARY KEY ("IdTimeSheetHour"),
```

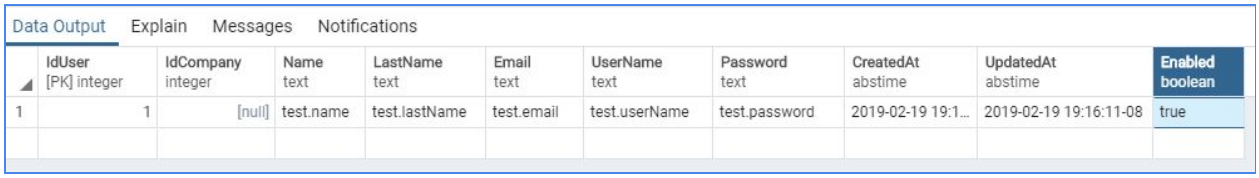
Once we re-run our create scripts, lets also re-run the test SQL INSERT script:

```
INSERT INTO public."User"(
    "Name", "LastName", "Email", "UserName", "Password", "CreatedAt",
    "UpdatedAt", "Enabled")
VALUES ('test.name', 'test.lastName', 'test.email',
        'test.userName', 'test.password', now(), now(), true);
```

We now receive the following output on our console:

```
INSERT 0 1

Query returned successfully in 73 msec.
```

Perfect, now we now that at least our User's table definition is operational as we had originally expected it to be. Lets execute a test query to ensure that our primary-key values are also coming back as expected.



Data Output | Explain | Messages | Notifications

	IdUser [PK] integer	IdCompany integer	Name text	LastName text	Email text	UserName text	Password text	CreatedAt abstime	UpdatedAt abstime	Enabled boolean
1	1	[null]	test.name	test.lastName	test.email	test.userName	test.password	2019-02-19 19:1...	2019-02-19 19:16:11-08	true

Excellent, the primary key value came back as 1 without us manually inserting it, as we were originally looking for. Now we can go on to our INSERT operations.

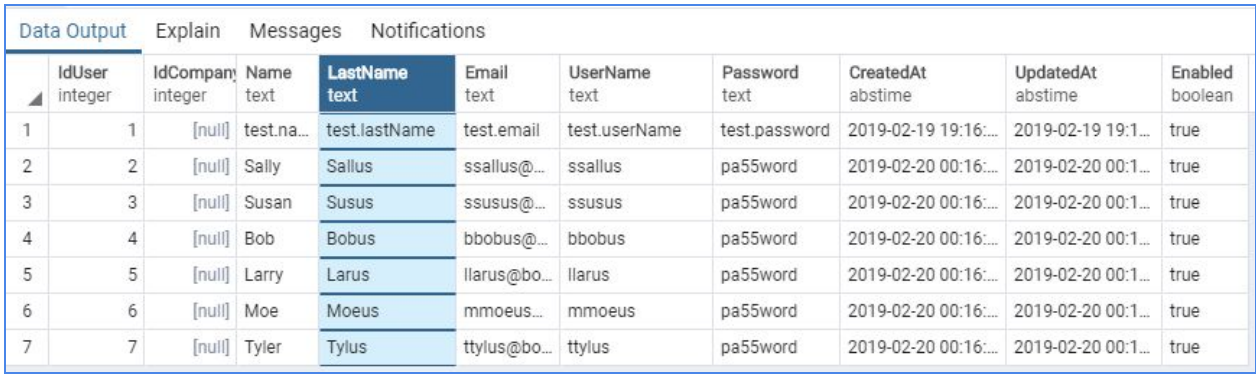
User Table Inserts

We have 6 users as we outlined in our Data Story section: Sally, Susan, Bob, Larry, Moe, Tyler. This script will create row-entries for each user.

```
-- Create script for: Sally, Susan, Bob, Larry, Moe, Tyler

-- Sally Sallus, ssallus@boingboing.com
INSERT INTO public."User"(
    "Name", "LastName", "Email", "UserName",
    "Password", "CreatedAt", "UpdatedAt", "Enabled")
VALUES (
    'Sally', 'Sallus', 'ssallus@boingboing.com',
    'ssallus', 'pa55word', now(), now(), true);

-- Susan Susus, ssusus@boingboing.com
INSERT INTO public."User"(
    "Name", "LastName", "Email", "UserName",
    "Password", "CreatedAt", "UpdatedAt", "Enabled")
VALUES (
    'Susan', 'Susus', 'ssusus@boingboing.com',
    'ssusus', 'pa55word', now(), now(), true);

-- Bob Bobus, bbobus@boingboing.com
INSERT INTO public."User"(
    "Name", "LastName", "Email", "UserName",
    "Password", "CreatedAt", "UpdatedAt", "Enabled")
VALUES (
    'Bob', 'Bobus', 'bbobus@boingboing.com',
    'bbobus', 'pa55word', now(), now(), true);

-- Larry, Larus, llarus@boingboing.com
INSERT INTO public."User"(
    "Name", "LastName", "Email", "UserName",
    "Password", "CreatedAt", "UpdatedAt", "Enabled")
VALUES (
    'Larry', 'Larus', 'llarus@boingboing.com',
    'llarus', 'pa55word', now(), now(), true);

-- Moe, Moeus, mmoeus@boingboing.com
INSERT INTO public."User"(
    "Name", "LastName", "Email", "UserName",
    "Password", "CreatedAt", "UpdatedAt", "Enabled")
VALUES (
    'Moe', 'Moeus', 'mmoeus@boingboing.com',
    'mmoeus', 'pa55word', now(), now(), true);
```

```
-- Tyler, Tylus, ttylus@boingboing.com, pa55word,
INSERT INTO public."User"(
    "Name", "LastName", "Email", "UserName",
    "Password", "CreatedAt", "UpdatedAt", "Enabled")
VALUES ('Tyler', 'Tylus', 'ttylus@boingboing.com',
    'ttylus', 'pa55word', now(), now(), true);
```

Now, let's verify that our users were all correctly created with another SELECT * operation.



Data Output | Explain | Messages | Notifications

	IdUser integer	IdCompany integer	Name text	LastName text	Email text	UserName text	Password text	CreatedAt abstime	UpdatedAt abstime	Enabled boolean
1	1	[null]	test.na...	test.lastName	test.email	test.userName	test.password	2019-02-19 19:16:...	2019-02-19 19:1...	true
2	2	[null]	Sally	Sallus	ssallus@...	ssallus	pa55word	2019-02-20 00:16:...	2019-02-20 00:1...	true
3	3	[null]	Susan	Susus	ssusus@...	ssusus	pa55word	2019-02-20 00:16:...	2019-02-20 00:1...	true
4	4	[null]	Bob	Bobus	bbobus@...	bbobus	pa55word	2019-02-20 00:16:...	2019-02-20 00:1...	true
5	5	[null]	Larry	Larus	llarus@bo...	llarus	pa55word	2019-02-20 00:16:...	2019-02-20 00:1...	true
6	6	[null]	Moe	Moeus	mmoeus...	mmoeus	pa55word	2019-02-20 00:16:...	2019-02-20 00:1...	true
7	7	[null]	Tyler	Tylus	ttylus@bo...	ttylus	pa55word	2019-02-20 00:16:...	2019-02-20 00:1...	true

Excellent, our users are all there just as we expected! Except, we now have one issue that, we need (or, should) delete our test user so, let's do that now.

```
-- DELETE FROM public."User"
-- SELECT * from public."User" WHERE "IdUser" = 1;
DELETE from public."User" WHERE "IdUser" = 1;
-- SELECT * from public."User" WHERE "IdUser" = 1;
```

Executing this script should produce the following results.

```
DELETE 1

Query returned successfully in 168 msec.
```

Within the delete script, there are two SQL scripts captured in the comments that can be used to determine first that the entry exists, so that we know that the WHERE clause in our delete is correct and, then it can be ran again to verify that the entry no longer exists by running it and seeing that it returns no results. If our SQL skills where a bit more advanced, we could use the SELECT to drive the input to the DELETE operation's WHERE clause, a technique that we can utilize later on when we begin doing more complex operations in PGSQL. Now, let's do the same things for the rest of our data instance items.

Native Update Operations

Native Delete Operations

Native Retrieval Operations

Clojure Data-Operation Expressions

Clojure Datamodel Expression

Clolure Insert Expression

Clojure Update Expression

Clojure Delete Expression

Clojure Retrieval Expression

MongoDB

TODO: We might not do or consider this one… we'll see… Hold tight Mongo! :)

MongoDB Overview

The MongoDB Datamodel

The MongoDB Native Query Language

Datomic

Datomic Overview

Product offerings and variations

Clojure Data-storage Components, Frameworks and Toolkits

TODO: This is where the libs (i.e hugsql, korma) will go...

Analysis

TOSAY: Everything has been building up to this - the reason and purpose for thie document, these exercises, everything prior - the moment we've all been waiting for! :)

Assessments on a standalone basis….

SQL vs Datalog - A 1-to-1 Compare

TODO: directly compare SQL syntax against Datomic Datalog

References and Resources

Bibliographical References

TODO: Include RDF Standard here
TODO: Include SPARQL References
TODO:
TODO: Include SQL Bibliographies

SQL-86 (or SQL-87) is the *ISO 9075:1987* standard of 1987

SQL-89 is the *ISO/IEC 9075:1989* standard of 1989

SQL-92 is the *ISO/IEC 9075:1992* standard of 1992

SQL:1999 is the *ISO/IEC 9075:1999* standard of 1999

SQL:2003 is the *ISO/IEC 9075:2003* standard of 2003

SQL:2006 is the *ISO/IEC 9075:2006* standard of 2006

SQL:2008 is the *ISO/IEC 9075:2008* standard of 2008

SQL:2011 is the *ISO/IEC 9075:2011* standard of 2011

SQL:2016 is the *ISO/IEC 9075:2016* standard of 2016

Web Resources

Web Articles

Specific to Datomic

Dustin's Datomic FAQ
http://www.dustingetz.com/:datomic-faq

Specific to Postgres

Jonathan Katz: WITH Queries: Present & Future
https://info.crunchydata.com/blog/with-queries-present-future-common-table-expressions

Bruce Momjian: Imperative to Declarative to Imperative

Baron Schwartz: Citus: Scale-Out Clustering and Sharding for PostgreSQL

Markus Winand: PostgreSQL 11 Reestablishes Window Functions Leadership

Jobin Augustine: plprofiler – Getting a Handy Tool for Profiling Your PL/pgSQL Code

Joe Conway: PostgreSQL Deep Dive: How Your Data Model Affects Storage

Bruce Momjian: Composite Values

Twitter

@datomic_team

@cognitect

@clojure_conj

@puredanger

@CursiveIDE

@stuarthalloway

@ClojureWerkz

Video Resources

Specific to Datomic

Rich Hickey on Datomic Ions, September 12, 2018
https://www.youtube.com/watch?v=thpzXjmYyGk
Length: 1:45:49
Notes/Thoughts:

KotlinConf 2018 - Datomic: The Most Innovative DB You've Never Heard Of
https://www.youtube.com/watch?v=hicQvxdKvnc
Length: 39:45
Notes/Thoughts:

Reactive Datalog for Datomic - Nikolas Göbel
https://www.youtube.com/watch?v=ZgqFlowyfTA
Length: 31:36
Notes/Thoughts:

Day of Datomic Cloud - Session 1
https://www.youtube.com/watch?v=yWdfhQ4_Yfw
Length: 41:28
Notes/Thoughts:

Rich Hickey: Deconstructing the Database
https://www.youtube.com/watch?v=Cym4TZwTCNU
Length: 1:06:23
Notes/Thoughts:

Datomic by Rich Hickey
https://www.youtube.com/watch?v=4iaIwiemqfo
Length: 1:30:55
Notes/Thoughts:

I put 7 years of meal data in Datomic - Here's what I learned - Christian Johansen
https://www.youtube.com/watch?v=ORQ2qs8GHsQ
49:27
Length:
Notes/Thoughts:

Intro to Datomic
https://www.youtube.com/watch?v=RKcqYZZ9RDY
Length:

Notes/Thoughts:

YOW! 2016 Stuart Sierra - Building Flexible Systems with Clojure and Datomic
https://www.youtube.com/watch?v=5YGcHIwDRao
Length:
Notes/Thoughts:

Lucas Cavalcanti & Edward Wible - Exploring four hidden superpowers of Datomic
https://www.youtube.com/watch?v=7Im3K8zVOdY
Length:
Notes/Thoughts:

Architecting a Modern Financial Institution
https://www.youtube.com/watch?v=VYuToviSx5Q
Length:
Notes/Thoughts:

Datomic: up and running
https://www.youtube.com/watch?v=ao7xEwCjrWQ
Length:
Notes/Thoughts:

Simplifying ETL with Clojure and Datomic - Stuart Halloway
https://www.youtube.com/watch?v=oOON--g1PyU
Length:
Notes/Thoughts:

David Greenberg - Building Interactive Query Tools on Datomic
https://www.youtube.com/watch?v=YHctJMUG8bI
Length:
Notes/Thoughts:

"Immutable Data Science with Datomic, Spark and Kafka" by Konrad Scorciapino and Mauro Lopes
https://www.youtube.com/watch?v=VexLSuOvb0w
Length:
Notes/Thoughts:

Lessons from 4 Years with Datomic w/ Robert Stuttaford
https://www.youtube.com/watch?v=35z96UzUCpg
Length:
Notes/Thoughts:

Day of Datomic Cloud - Session 2
https://www.youtube.com/watch?v=ZP-E2IgqKfA
Length:
Notes/Thoughts:

Datomic Made Easy Datomic in the Cloud - Stuart Halloway
https://www.youtube.com/watch?v=Ljvhjei3tWU
Length:
Notes/Thoughts:

WormBase database migration to Datomic on AWS: A case Study - Adam Wright
https://www.youtube.com/watch?v=RzE9iencxT0
Length:
Notes/Thoughts:

PolyConf 16: Datomic in production, 1 year in / Hans Hübner
https://www.youtube.com/watch?v=0y6QK813new
Length:
Notes/Thoughts:

Working with Datomic
https://www.youtube.com/watch?v=kd1yTmx7m2A
Length:
Notes/Thoughts:

Serverless-ish: Zero to App with Datomic Cloud and GraphQL - Chris Johnson Bidler
https://www.youtube.com/watch?v=SUfgKiTI8nE
Length:
Notes/Thoughts:

Specific to MongoDB

Specific to PostgreSQL

Specific to Clojure

TODO: remove this section

The Language of the System - Rich Hickey
https://www.youtube.com/watch?v=ROor6_NGIWU

Expert to Expert: Rich Hickey and Brian Beckman - Inside Clojure
https://www.youtube.com/watch?v=wASCH_gPnDw
Length:
Notes/Thoughts: