Matthew Dirks

December 9, 2020

Term Project

MET CS 669

Prof. Warren Mansur

Section O2, Group 3

Contents

Project Direction Overview

Provide an overview that describes who the database will be for, what kind of data it will contain, how you envision it will be used, and most importantly, why you are interested in it.

Overview of Nugatoria

Nugatoria[1] will be an app for organizing pages of notes into Notebooks. It will be similar to Microsoft OneNote but with different priorities: the primary use case is a single user storing notes and accessing notebooks across different platforms. Hence where OneNote allows for very rich formatting and concurrent editing, Nugatoria will focus on storing pages in a simple format and on reliable version control and conflict resolution features. It will support several users sharing a notebook even though robust concurrent modification is not the primary intended use.

At a high level, Nugatoria will allow multiple **users**. Each user can have access to many **notebooks** (and may have different **permissions** for each), and a notebook may be accessed by many users. A notebook is a collection of **sections** and **section groups**. (A section may be contained in a section group or may be contained in the notebook.) A section group a collection of sections, which in turn is a collection of **pages**. Each page is a collection of **revisions.** (That is, one notebook can have one or many sections; one notebook can also have one or many section groups; one section can have one or many pages; one page can have one or many revisions). A revision may also have 0 or many **hashtags**, and a hashtag may point to one or many revisions.

Notebooks, section groups, sections, and pages are all examples of **NugatoriaContainers**. It is possible that, as this application grows, the specific NugatoriaContainers it supports will change over time.

I am interested in this project because I use Microsoft OneNote frequently and find it to be one of the most useful tools in my day-to-day life, but I am often frustrated by issues around slow syncing and page conflicts. I frequently edit pages on my phone and later on my computer, only to find that one of those versions did not sync properly. OneNote has some functionality for saving previous revisions but this feature often seems unreliable. OneNote also has many features that I rarely use; mostly I am interested in storing and accessing plain text, formatted lists, links, and images. I therefore want to replicate the features I find most useful, streamline my app by limiting feature creep, and focus on frequent versioning and the ability for users to manually address conflict resolution for different page versions. OneNote also does not have any native hashtag functionality, and it seems to me that it would be useful to be able to search through pages labeled with hashtags rather than purely hierarchically.

[1] The name *Nugatoria* is a Latin word meaning “trifles.” Specifically, it is the neuter plural nominative/accusative form of *nugatorius*; see the entry for that form in (e.g.) *A Latin Dictionary* (Lewis & Short, 1870)

Use Cases

Provide use cases that enumerate steps of how the database will be typically used, also identify significant database fields needed to support the use case.

We first give a few use cases and then, in the subsequent section, give the tables suggested by these use cases.

Use Case: User creates an account

1. User visits Nugatoria website and chooses “Create an account”
2. User enters first name, last name, username, email, and password.
3. Application creates a randomly-generated and unique salt string
4. Application stores the first name, last name, username, salt string, and email in the database, along with the result of running the password and salt string through a hash function.
5. Application creates default NugatoriaContainers and stores them in the database. Specifically, it creates a Notebook titled “My Notebook” containing a default section (“Untitled Section”) which contains a default page (“Untitled Page”) which in turn contains an empty revision.
6. Application updates permissions in the database so that the user has Owner-level permission for “My Notebook” and its member containers
7. User is shown the default NugatoriaContainers.

Use Case: User resets password

1. User clicks “Forgot password” link
2. User enters email address
3. Application sends email to user with link to reset password
4. User chooses new password
5. Application generates new salt string and calculates hashed password
6. Application stores new salt and hashed password in database.

Use Case: User reverts to earlier page version

1. User right-clicks the name of a “page”
2. Contextual menu appears
3. User selects “Page revisions” option
4. Interface displays recent page revisions (perhaps the five most recent, with an option to view more). Each revision is labeled with a timestamp and the name of the user who made the edit (either in “Lastname, Firstname” format or using the editor’s custom “signature”)
5. User selects an earlier revision
6. Contents of earlier revision are displayed.
7. User clicks on page to begin editing it.
8. Application duplicates the revision and stores it in the database as the newest revision before any edits are committed.

Use Case: Conflict Resolution

1. User edits page on phone (thus triggering the application to create a new revision)
2. Because of connectivity issues, those edits are not stored in the database immediately
3. User edits an out-of-date version of page on computer
4. Edits from earlier are stored (after the conflicting edit has been stored)
5. Application recognizes that there is a conflict
6. Application notifies user of conflict (error message)
7. Application displays two conflicting revisions
8. User selects the revision to use
9. Application duplicates the selected revision and sets it as the “latest” revision.

10. Application adds a note to the non-selected revision marking it as “deprecated”. (However, the user may still access it as though it were any other revision).

Use Case: Moving or Copying a Page

A page may be moved or copied from one section to another. Analogously, a section may be moved or copied from one section group to another section group, and a section may be moved or copied from one notebook to another. However, there is one important exception to the general pattern: a revision belongs permanently to a page and may not be moved from one page to another.

1. User right-clicks on a NugatoriaContainer name (e.g. a page name).
2. Application displays contextual menu. If the NugatoriaContainer is movable (the attribute is_movable is set to true (1)), this menu contains the choice “Move” and “Copy”
3. User selects “Copy”
4. User right-clicks within a section (often a different section, but not necessarily).
5. Application displays contextual menu, which includes a “Paste” option
6. User selects “Paste”
7. Application creates a new NugatoriaContainer, identical to the first but with a different row in the database and with the appropriate new owner_id (if applicable). If the user has pasted a copy of this section in the same NugatoriaContainer, the string “ (Copy)” (with leading whitespace) is appended to the end of the name.
8. The application displays the new NugatoriaContainer in the user interface.

Use Case: User A shares notebook with User B

1. User A (Alice) clicks “Share this notebook”
2. Interface prompts user to enter the email address of another user and select the desired permission level for that user (eg “Read only”, “Read and Edit”, “Read, Edit, and Share”)
3. Alice enters the email of User B (Bob)
4. Application creates an Invitation referencing inviter Alice, invitee Bob, the container_id of the notebook, and the permission_level_id of the relevant permission level.
5. Application sends an email to Bob inviting him to the notebook
6. Bob accepts the invitation
7. If Bob does not have an account, he creates an account (see that use case, above)
8. Application creates a permission granting Bob the correct level of access.

Use Case: Add and remove hashtags to a revision

A page revision is comprised of fields that can contain various types of data (html, embedded file, embedded image, plain text, LaTeX code, hashtags). Those details will mostly be left to application logic for implementation. However, we will store hashtags in our database so that users can quickly find related pages.

1. User creates a page called “Mad Scribbles” (and therefore a new revision) and types notes in a plain text field
2. User adds a “hashtag” field and adds three hashtags: #ramblings #rantings #ravings
3. Application commits the changes
4. User edits the page again; application creates a new revision
5. User removes the hashtag #ravings.
6. Application updates the TaggedRevisions table to reflect that the tag has been deprecated for this page
7. User clicks on the #ramblings tag.
8. Application displays a list of pages with the #ramblings tag, including “Mad Scribbles,” and a list of related hashtags, including #rantings and #ravings
9. User clicks on #ravings
10. Application displays a list of pages with the #ravings tag. “Mad Scribbles” is not displayed on the main list, but is displayed to the side on a list labeled “Previously Tagged #ravings”

Tables and Attributes

Table: Account

Field	Data type	Constraints	What it stores	Why it's needed
account_id (PRIMARY KEY)	DECIMAL(12)	UNIQUE NOT NULL AUTO_INCREMENT	Unique integer to identify accounts	Users might want to change their name, email, signature, or password. Furthermore, we might have several users with the same name, signature, etc. An id number allows us to identify each account unambiguously regardless of changes.
last_name	VARCHAR(1024)		Last name of the user	Useful for displaying user name in formal settings ("Page Revision by Matthew Dirks at 11/08/20, 2:57 PM")
first_name	VARCHAR(1024)	NOT NULL	First name of the user	Useful for displaying friendly, informal messages to user ("Hello, Matthew!")
signature	VARCHAR(32)	NOT NULL	Short string used to identify which user has made changes to a page (e.g. the user's initials)	Allows users to specify how collaborators recognize their work. For example, if I modify part of a page, I might choose to have my initials ("mjd") appear next to the edit.
username	VARCHAR(32)	UNIQUE NOT NULL	Username for login	
email	VARCHAR(1024)	NOT NULL	User's email	Necessary for password reset process
salt_str	VARCHAR(1024)	NOT NULL UNIQUE	A string, randomly and securely generated when the account is created and re-generated if the user resets their password	Different users who have the same password ("asdfghj") should nevertheless not have the same hash; adding salt allows randomness and hence security.
hashed_password	VARCHAR(1024)	NOT NULL	The result of running the concatenation of the user's password and the salt string through a hash function	We do not want to store passwords in plaintext, so instead we store a hashed version.
account_balance	DECIMAL(12, 2)		The amount of credit or debt a user has	Used for billing; although Nugatoria will initially be a free app, we may eventually wish to charge for it and let

				users pay a fee at regular intervals.
monthly_charge	DECIMAL(12, 2)		The amount a user is billed each month	

Table: AccountBalanceChange

Field	Data type	Constraints	Description
change_id (PRIMARY KEY)	DECIMAL(12)	UNIQUE NOT NULL	Primary key for referencing the change
account_id (FOREIGN KEY REFERNCES Account(account_id))	DECIMAL(12)	NOT NULL	Foreign key to the account table, referencing the account whose balance changed
old_balance	DECIMAL(12, 2)	NOT NULL	The account balance before the change
new_balance	DECIMAL(12, 2)	NOT NULL	The account balance after the change
change_date	DATE	NOT NULL	The date of the change

Table: NugatoriaContainer

Because Oracle reserves the use of the word Container, our table is called NugatoriaContainer. However, in this document the word 'container' may also be used passim.

Field	Data type	Constraints	What it stores	Why it's needed
container_id (PRIMARY KEY)	DECIMAL(12)	UNIQUE NOT NULL AUTO_INCREMENT	Unique integer to identify container types	Container type names may change, so we need an unambiguous way to refer to them.
container_type_id (FOREIGN KEY REFERENCES ContainerTypes (container_type_id))	DECIMAL(12)	NOT NULL	Integer specifying the type of container	Allow the interface to correctly display the container
container_name	VARCHAR(64)	NOT NULL	Name of container (e.g. "My Notebook," or "Lists", or "Shopping List")	Provide a human-friendly reference to a container (notebook, section group, section, revision, etc.)
owner_id	DECIMAL(12)		The id of the container that contains this container (if relevant). For example, if this record refers to a section, the owner_id may refer to a section group. This may be null (e.g. a notebook is not contained in a higher level)	Allows for a recursive relationship between two containers representing the hierarchical nature of the notebook structure.
is_movable	INTEGER	NOT NULL	1 (true) if a container may be moved or copied from one owner to another, 0 (false) otherwise. For example, a page may be moved or copied from one section to another section, but a revision may not be moved or copied from one page to another page.	Records whether a container may be moved or copied from one owner to another.

Table: ContainerType

Field	Data type	Constraints	What it stores	Why it's needed
container_type_id (PRIMARY KEY)	DECIMAL(12)	UNIQUE NOT NULL AUTO_INCREMENT	Unique integer to identify container types	Container type names may change, so we need an unambiguous way to refer to them.
container_type	VARCHAR(32)	NOT NULL	Names of container types (e.g. 'Notebook', 'Section group', 'Section', 'Page', 'Revision')	Human-friendly reference

Table: Notebook

Field	Data type	Constraints	What it stores	Why it's needed
container_id (PRIMARY KEY, FOREIGN KEY REFERENCES CONTAINER (container_id))	DECIMAL(12)	UNIQUE NOT NULL	Reference to the container supertype	Primary/Foreign key in specialization-generalization relationship
time_created	TIMESTAMP	NOT NULL	Time at which notebook was created	Useful record
last_sync	TIMESTAMP	NOT NULL	Time of last sync	Useful for flagging conflict resolutions
owning_account (FOREIGN KEY REFERENCES ACCOUNT (account_id))	DECIMAL(12)		account_id for account that initially created notebook, if applicable	When a new account is created, a new notebook is also created. This field allows the database to match the new account with the new notebook.

Table: Revision

When a user views a page, they are really viewing a revision. When they update the page, a duplicate is created and becomes the latest revision. The previous revisions are stored for possible future access.

Field	Data type	Constraints	What it stores	Why it's needed
container_id (PRIMARY KEY, FOREIGN KEY REFERENCES CONTAINERS (container_id))	DECIMAL(12)	UNIQUE NOT NULL	Reference to the container supertype	Primary/Foreign key in specialization-generalization relationship
location	VARCHAR(4000)	UNIQUE NOT NULL	A reference to the location where the revision data is stored (e.g. the URL or local address of a data file)	A revision is the "lowest level" of our container hierarchy and contains user-entered data (e.g. notes and lists). That data will be saved in a data file somewhere; this field tells Nugatoria where to look.
time_saved	TIMESTAMP	NOT NULL	The date and time at which a revision is created	The revision history will display the time and date when each revision was created.
deprecated	INTEGER	NOT NULL DEFAULT 0	0 (False) or 1 (True); indicating whether the	

			revision has been deprecated through conflict resolution. (This would be a BOOLEAN type if Oracle supported that type).	
predecessor_id FOREIGN KEY REFERENCES REVISION(container_id)	DECIMAL(12)		id of the previous revision, if there is one	Useful in determining whether an edit conflict must be resolved. We hope that these values are unique, but do not require it. When two edits have the same predecessor, they conflict. Hence we do not use the UNIQUE constraint because doing so would make it difficult to identify conflicts.

Table: NugatoriaPermission

As with Container, Oracle reserves the word Permission; hence this table is called NugatoriaPermission, but the word 'Permission' is used interchangeably passim in this document.

In the initial design of this application, permission is granted at the notebook level. For example, we do not allow the case where a user has "Read" access to an entire notebook but "Read and Edit" access to an individual page. However, in our design we grant permission at the container level in case we wish to add that functionality in the future.

Composite Primary Key: (user_id, container_id)				
Field	**Data type**	**Constraints**	**What it stores**	**Why it's needed**
account_id (FOREIGN KEY REFERENCES Account (account_id))	DECIMAL(12)	NOT NULL	Reference to the id of the user whose permission level is being stored	Unambiguous reference to user
container_id (FOREIGN KEY REFERENCES Containers (container_id))	DECIMAL(12)	NOT NULL	The container (e.g. notebook) for which this user has been granted an access permission	Unambiguous reference to container
permission_level_id (FOREIGN KEY REFERENCES PermissionLevels (permission_level_id))	DECIMAL(12)	NOT NULL	The permission level at which the user has access to the container (e.g. "Read Only")	Specifies the type of permission that the user has

Table: PermissionLevel

This table stores different permission levels (eg "Read only", "Read and Edit", "Read, Edit, and Share")

Field	Data type	Constraints	What it stores	Why it's needed
permission_level_id (PRIMARY KEY)	DECIMAL(12)	UNIQUE NOT NULL AUTO_INCREMENT	Unique integer to identify permission levels	Unambiguous reference
permission_description	VARCHAR(1024)	NOT NULL	Human-readable description of permission level (eg "Read only", "Read and Edit", "Read, Edit, and Share")	Human-readable reference

Table: Invitation

Field	Data type	Constraints	What it stores	Why it's needed
invitation_id (PRIMARY KEY)	DECIMAL(12)	UNIQUE NOT NULL AUTO_INCREMENT	Synthetic key for invitations	Unambiguous reference
permission_level_id (FOREIGN KEY REFERENCES PermissionLevel)	DECIMAL(12)	NOT NULL	The permission level being granted	The invitee is granted access at a specific permission level
inviter_id (FOREIGN KEY REFERNCES Accounts)	DECIMAL(12)	NOT NULL	Primary key referring to the user sending the invitation	Possibly useful for historical audits
invitee_id (FOREIGN KEY REFERNCES Accounts)	DECIMAL(12)	NOT NULL	Primary key referring to the user receiving the invitation	Permission must be granted to a specific user
container_id (FOREIGN KEY REFERENCES Container)	DECIMAL(12)	NOT NULL	A reference to the container to which permission is being granted	Unambiguous reference
time_sent	TIMESTAMP		Time at which the invitation is created	Possibly useful for historical audits

Table: SentInvitation

Composite Primary Key: (invitation_id, sender_id)				
Field	**Data type**	**Constraints**	**What it stores**	**Why it's needed**
invitation_id (PRIMARY KEY) (FOREIGN KEY REFERENCES Invitation(invitation_id))	DECIMAL(12)	NOT NULL	id of the invitation in supertype	Primary/Foreign key in specialization-generalization relationship
sender_id (PRIMARY KEY) (FOREIGN KEY REFERENCES ACCOUNT (account_id))	DECIMAL(12)	NOT NULL	id of user who sends the invitation	
time_sent	TIMESTAMP	NOT NULL	time at which invitation was sent	Historical purposes

Table: ReceivedInvitation

Composite Primary Key: (invitation_id, recipient_id)				
Field	**Data type**	**Constraints**	**What it stores**	**Why it's needed**
invitation_id (PRIMARY KEY) (FOREIGN KEY REFERENCES Invitation(invitation_id))	DECIMAL(12)	NOT NULL	id of the invitation in supertype	Primary/Foreign key in specialization-generalization relationship
recipient_id (PRIMARY KEY) (FOREIGN KEY REFERENCES ACCOUNT (account_id))	DECIMAL(12)	NOT NULL	id of user who sends the invitation	
accepted	BOOLEAN	NOT NULL	Boolean (0 = true, 1 = false) encoding whether the invitee has accepted the invitation	Potentially useful for troubleshooting (e.g. if a user cannot figure out why they cannot access a notebook, it may be because they did not accept the invitation; this

				attribute could help clarify that situation).

Table: Hashtag

Field	Data type	Constraints	What it stores	Why it's needed
hashtag_id	DECIMAL(12)	UNIQUE NOT NULL AUTO_INCREMENT	Reference to the id of the hashtag	Unambiguous reference to hashtag
label	VARCHAR(32)	NOT NULL	The hashtag itself (without the leading '#' character; eg 'Ramblings')	Searchable, human-readable

Table: ContainerTagBridge

Composite Primary Key: (hashtag_id, revision_id)				
Field	**Data type**	**Constraints**	**What it stores**	**Why it's needed**
hashtag_id (FOREIGN KEY REFERENCES Hashtag(hashtag_id))	DECIMAL(12)	NOT NULL	Reference to the id of the hashtag	These fields together store the fact that a given container has been tagged with a given hashtag
container_id (FOREIGN KEY REFERENCES Revision(container_id))	DECIMAL(12)	NOT NULL	Reference to the id of a container (usually a page) using that hashtag	
deprecated	INTEGER (viewed as a BOOLEAN)	NOT NULL DEFAULT 0	0 (False) if the most recent revision contains this tag; 1 (True) if this tag has been deleted.	Allows user to see pages that once used this tag but no longer do.

Structural Database Rules

Container rules.

On the application side, the Container class will have many subclasses: Notebook, Section Group, Section, Page, and Revision. From the relational database perspective, there is no need for separate subtypes and tables for most of these; rather, they are all stored as "Containers." However, Revisions and Notebooks have unique attributes not shared by other containers. Hence we have the structural database rule: *a container may be a notebook, or a revision, or neither.*

This is a disjoint and partially complete specialization-generalization relationship.

- A container is specified by one ContainerType; a ContainerType may specify many containers.
- A container ("owner") may own many containers ("members"). A container ("member") may be owned by at most one container ("owner"). We may look at specific examples of this rule:
    - At the notebook level:
        - A notebook has no owner.
        - A notebook may own one or many section groups; a section group is owned by one notebook.
        - A notebook may own one or many sections; a section may be owned by one notebook.
        - A notebook owns at least one container (either a section or a section group).
    - At the section group level:
        - A section group owns one or many sections; a section may be owned by at most one section group.
    - At the section level:
        - A section is owned by one container (either a notebook or a section group).
            - The other directions of this business rule, that a notebook may own one or many sections and that a section group does own one or many sections, are listed above.

        - A section owns one or many pages; a page is owned by one section.
    - At the page level:
        - A page owns one or many revisions; a revision is owned by one page.
    - At the revision level:
        - A revision may be preceded by one revision; a revision may precede another revision. (The first revision of a page has no predecessor; the most recent revision does not precede another.)

User Permission Rules

A *permission* is a statement such as "A user has read-only access to the notebook 'Scott Joplin Analyses'".

A *user* is represented in our database by an *account*, and I use the words here interchangeably.

- A user may hold many permissions (although at most one for a given container); a permission is held by one user.
- A permission references one container; a container is referenced by at least one permission. (If there are no users with at least some access to a container, the container would be permanently inaccessible!)
- A permission is specified by one permission level; a permission level may specify many permissions.
- A user may send many SentInvitations. A SentInvitation is sent by one user.
- An ReceivedInvitation is received by one user; one user may receive many ReceivedInvitations.
- Every invitation is a SentInvitation, a ReceivedInvitation, or both.
- An invitation references one permission level; a permission level may be referenced by many invitations.
- An invitation references one container; a container may be referenced by many invitations.

Other Rules

- A revision may reference many hashtags; a hashtag may be referenced by many revisions.
- An account may be referenced by an AccountBalanceChange; every AccountBalanceChange references one account.

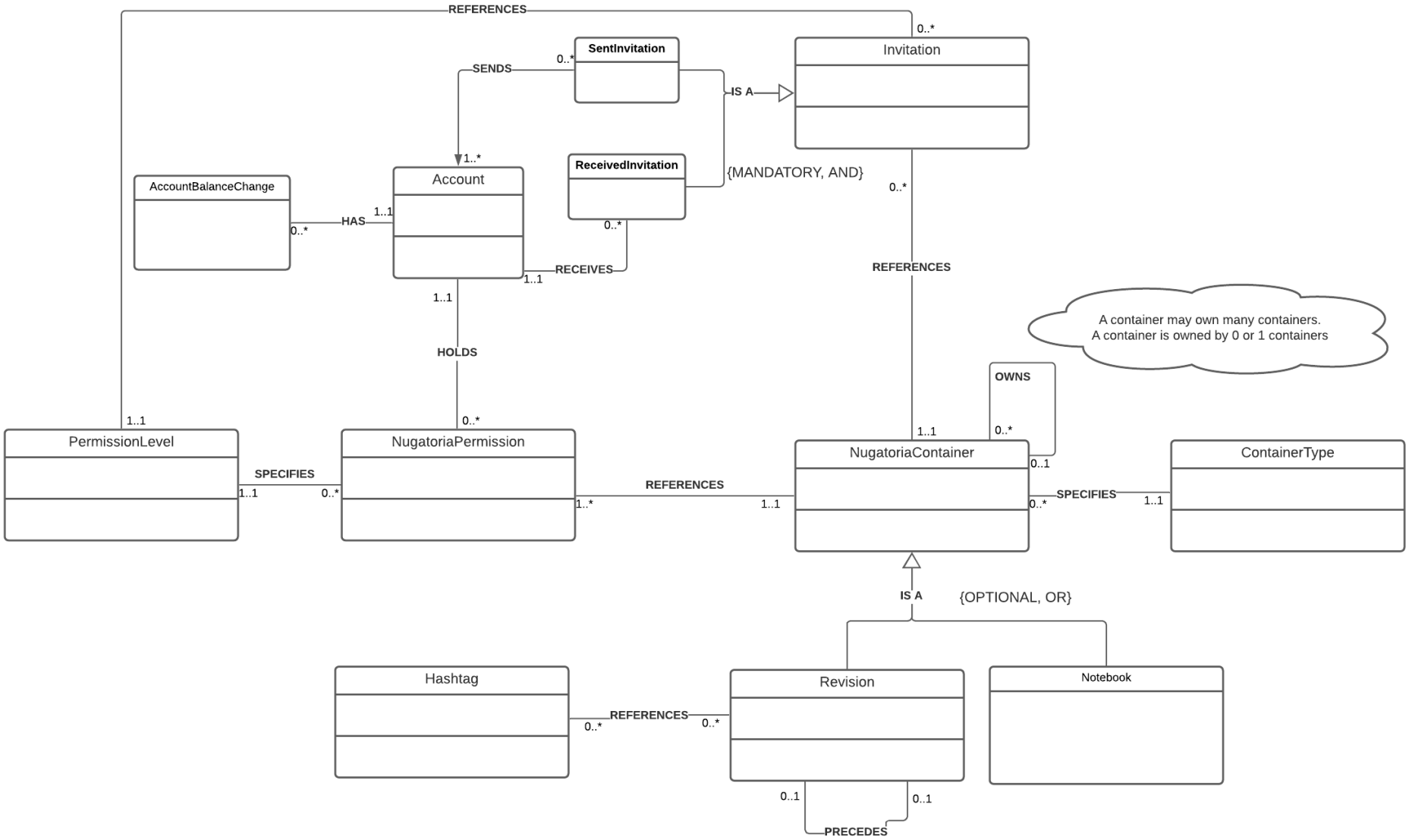
Summary of Business Rules

The business rules outlined above may be restated more concisely:

- A container is specified by one ContainerType; a ContainerType may specify many containers.
- A container may own many containers; a container may be owned by at most one container.
- A container may own many revisions; a revision is owned by one container.
- A container may be a notebook, a revision, or neither.
- An account may hold many permissions; a permission is held by one account.
- A permission references one container; a container is referenced by one or many permission
- A permission is specified by one permission level; a permission level may specify many permissions.
- A user may send many SentInvitations. A SentInvitation is sent by one user.
- An ReceivedInvitation is received by one user; one user may receive many ReceivedInvitations.
- Every invitation is a SentInvitation, a ReceivedInvitation, or both.
- An invitation references one permission level; a permission level may be referenced by many invitations.
- An invitation references one container; a container may be referenced by many invitations.
- A may reference many hashtags; a hashtag may be referenced by many revisions.
- A revision may be preceded by one revision; a revision may precede another revision.
- An account may be referenced by an AccountBalanceChange; every AccountBalanceChange references one account.

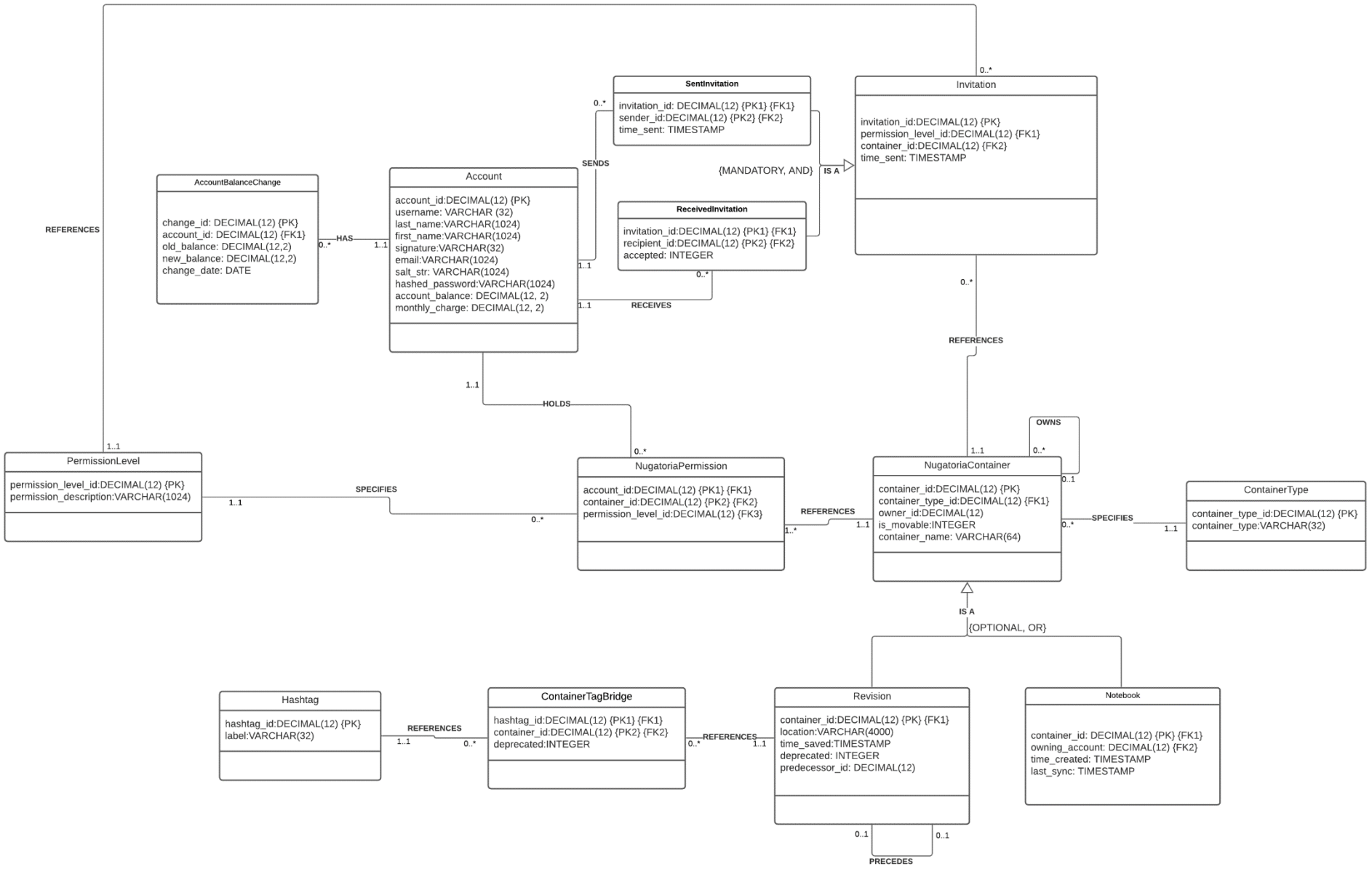
Conceptual Entity-Relationship Diagram

Below is an exported image of the conceptual entity-relationship diagram. It may be inspected more closely on LucidChart. The diagram contains the specialization-relationship that a Revision is a Container.



Physical Entity-Relationship Diagram

Below is an exported image of the physical entity-relationship diagram. It may be inspected more closely on LucidChart.



Normalization

We examine each of the entities in the Physical ERD in turn to ensure that each is in Boyce-Codd Normal Form.

- **Account:** This entity is in a table format with no repeating groups. Since the primary key is not composite, there can be no partial dependencies. This entity has several candidate keys: username and salt_str are both unique. Hence we have dependencies of the form A -> B -> C where B is not a primary key (e.g. account_id -> username -> email). However, since these dependencies involve attributes that are candidate keys, the entity is nevertheless in BCNF.
- **Invitation**: This entity is in a table format with no repeating groups. Since the primary key is not composite, there can be no partial dependencies. There are no transitive dependencies. The primary key is the only determinant (since the timestamp is not necessarily unique). Hence this entity is in BCNF.
- **SentInvitation**: This entity has two fields, both of which are components of the composite primary key; hence there are no partial dependencies or transitive dependencies and the primary key is the only determinant, so the entity is in BCNF.
- **ReceivedInvitation**: The structure of this entity is identical to that of SentInvitation, and the entity is in BCNF for the same reasons.
- **NugatoriaPermission**: This entity is in a table format with no repeating groups. The permission level (referenced by permission_level_id) depends on both account_id and container_id (a user has a specific permission level for a specific container), so there are no partial dependencies. There are no transitive dependencies since permission_level_id depends only on prime attributes. Since permission_level_id is not a determinant, only the primary key is determinant. Hence the entity is in BCNF.
- **PermissionLevel**: This entity is in table format with no repeating groups. The primary key is not composite so there are no partial entities. There is only one nonkey attribute, so there are no transitive dependencies. The only determinant is the primary key, so this entity is in BCNF.
- **ContainerType**: The structure of this entity is identical to that of PermissionLevel, and the entity is in BCNF for the same reasons.
- **NugatoriaContainer**: This entity is in table format with no repeating groups. The primary key is not composite so there are no partial entities. There are no transitive dependencies since none of container_type_id, owner_id, and is_movable depend on any other. For the same reason, container_id (the primary key) is the only determinant, so this entity is in BCNF.
- **Revision**: This entity is in table format with no repeating groups. The primary key is not composite so there are no partial entities. The attributes location, time_saved, deprecated, and predecessor_id are independent of each other; none of them is dependent on any other. Hence there are no transitive dependencies and the only determinant is the primary key, so this entity is in BCNF.
- **Notebook**: This entity is in table format with no repeating groups. The primary key is not composite so there are no partial entities. The only determinant is the primary key, since time_created and last_sync do not depend on each other. Hence the entity is in BCNF.
- **ContainerTagBridge**: This entity is in table format with no repeating groups. We have a composite primary key (containing the attributes *hashtag_id* and *container_id*), and the attribute *deprecated* depends on both of its attributes since a given hashtag is deprecated with reference to a specific container (a hashtag itself is not deprecated, nor is a container deprecated, but the conjunction of the two can be deprecated). Hence there are no partial dependencies. There are no transitive dependencies since the only determinants are key attributes. Since every determinant is part of the primary key, this entity is in BCNF.
- **Hashtag:** The structure of this entity is identical to that of ContainerType and PermissionLevel and the entity is in BCNF for the same reasons.

Minimum Entity Check: Nugatoria has 12 entities after normalization. If we do not count subtypes and count only supertypes (i.e exclude SentInviation, ReceivedInvitation, Revision, and Notebook from our count), it still has 8 entities. By both metrics I belive Nugatoria meets the project's complexity requirements.

Indexing the Database

Primary Key Indexes

The following primary keys will be indexed automatically by the RDBMS. These are all necessarily unique.

Invitation.invitation_id

SentInviation.invitation_id

SentInvitation.sender_id

ReceivedInvitation.invitation_id

ReceivedInvitation.recipient_id

Account.account_id

NugatoriaPermission.account_id

NugatoriaPermission.container_id

PermissionLevel.permission_level_id

NugatoriaContainer.container_id

ContainerType.container_type_id

Notebook.container_id

Revision.container_id

ContainerTagBridge.hashtag_id

ContainerTagBridge.container_id

Hashtag.hashtag_id

Foreign Key Indexes

We also create indexes on each foreign key. Note that foreign keys which are part of primary keys (eg the primary keys of subtype entities) are not included below since we have accounted for them already.

Column	Unique?	Description
Invitation.permission_level_id	Not unique	This foreign key references the PermissionLevel table. It is not unique because many invitations may be sent or received at any given invitation level
Invitation.container_id	Not unique	This foreign key references the NugatoriaContainer table. It is not unique because several users may have permission to access the same notebook.
NugatoriaPermission.permission_level_id	Not unique	This foreign key references the PermissionLevel table. We expect that many permissions will be granted at any given level. For example, any account that creates a notebook will initially have ReadWriteShare permission for that notebook.
NugatoriaContainer.container_type_id	Not unique	This foreign key references the ContainerType table. Many containers in the database will be of the same type, so this index is not unique.

Notebook.owning_account	Not unique	When a new account is created, a new notebook is also created; this field stores the account_id of the associated account so that permission can be granted to that account. Since one account may create many notebooks, this is not unique.
AccountBalanceChange.account_id	Not Unique	This foreign key references the Account table. An account may change many times, and each change will be reflected by a row in this table, so the account_id is not unique.

Query-Based Indexes

Finally, we consider query-based indexes.

1. One common query will be finding all the containers that belong to a given container. For example, our application will frequently need to access all sections and section groups belonging to a notebook or all revisions belonging to a page. The following query would retrieve that information:

```
SELECT Member.container_id AS member_id, Member.container_name AS member_name,
Owner.container_name AS owner_name
FROM NugatoriaContainer Member
JOIN NugatoriaContainer Owner ON Owner.container_id = Member.owner_id
WHERE Owner.container_id = 1;
```

Based on this query, we should index `NugaliaContainer.owner_id`. It is used in a frequent JOIN statement, is not usually null, and will sharply limit the rows retrieved from the NugatoriaContainer table. This index is not unique since many containers may be owned by the same container.

2. A second query will be useful in identifying edit conflicts. An edit conflict occurs if the most recent revision has the same predecessor as an earlier revision; this suggests that the earlier revision was not properly stored in the database (eg due to a bad internet connection). The following query is an example; it identifies all edit conflicts for a page with container_id = 3.

```
--1) Find all revisions belonging to a page
CREATE VIEW RevisionsOfPage3 AS
SELECT container_id, time_saved, predecessor_id
FROM NugatoriaContainer
NATURAL JOIN Revision
WHERE NugatoriaContainer.owner_id = 3 -- belonging to page 3
ORDER BY time_saved DESC;
--2) Find the most recent revision
CREATE VIEW MostRecentRevisionOfP3 AS
SELECT * from RevisionsOfPage3
Where RevisionsOfPage3.time_saved = (SELECT MAX(time_saved)
FROM RevisionsOfPage3);
--3) Find all other revisions with the same predecessor_id as the most recent
```

```
SELECT OwningPage.container_name, ConflictingRevision.container_id AS
id_of_conflicting_revision, ConflictingRevision.time_saved AS older_save,
ConflictingRevision.predecessor_id

FROM NugatoriaContainer ConflictingContainer

JOIN Revision ConflictingRevision ON ConflictingContainer.container_id =
ConflictingRevision.container_id

JOIN NugatoriaContainer OwningPage on ConflictingContainer.owner_id =
OwningPage.container_id

WHERE ConflictingContainer.owner_id = 3 --belonging to Page 3

--Not the most recent revision

AND ConflictingRevision.time_saved != (SELECT time_saved FROM MostRecentRevisionOfP3)

--and predecessor_id is same as that of most recent

AND ConflictingRevision.predecessor_id = (SELECT predecessor_id FROM
MostRecentRevisionOfP3);
```

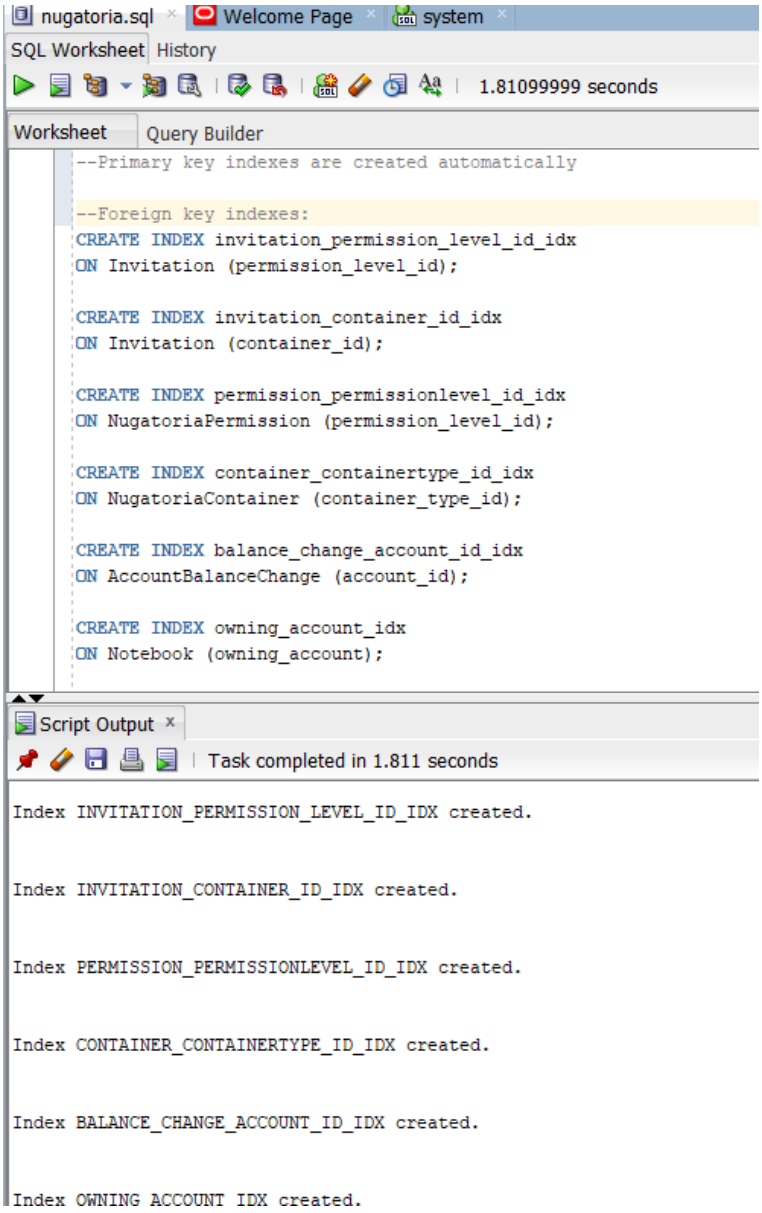
We have already noted that we should index `NugaliaContainer.owner_id`, and this query provides further justification. Additionally we note that `Revision.predecessor_id` and `Revision.time_saved` are used in WHERE statements, so we index those two columns. There is no expectation that either of those be unique (though `Revision.time_saved` may be), so we create a non-unique index.

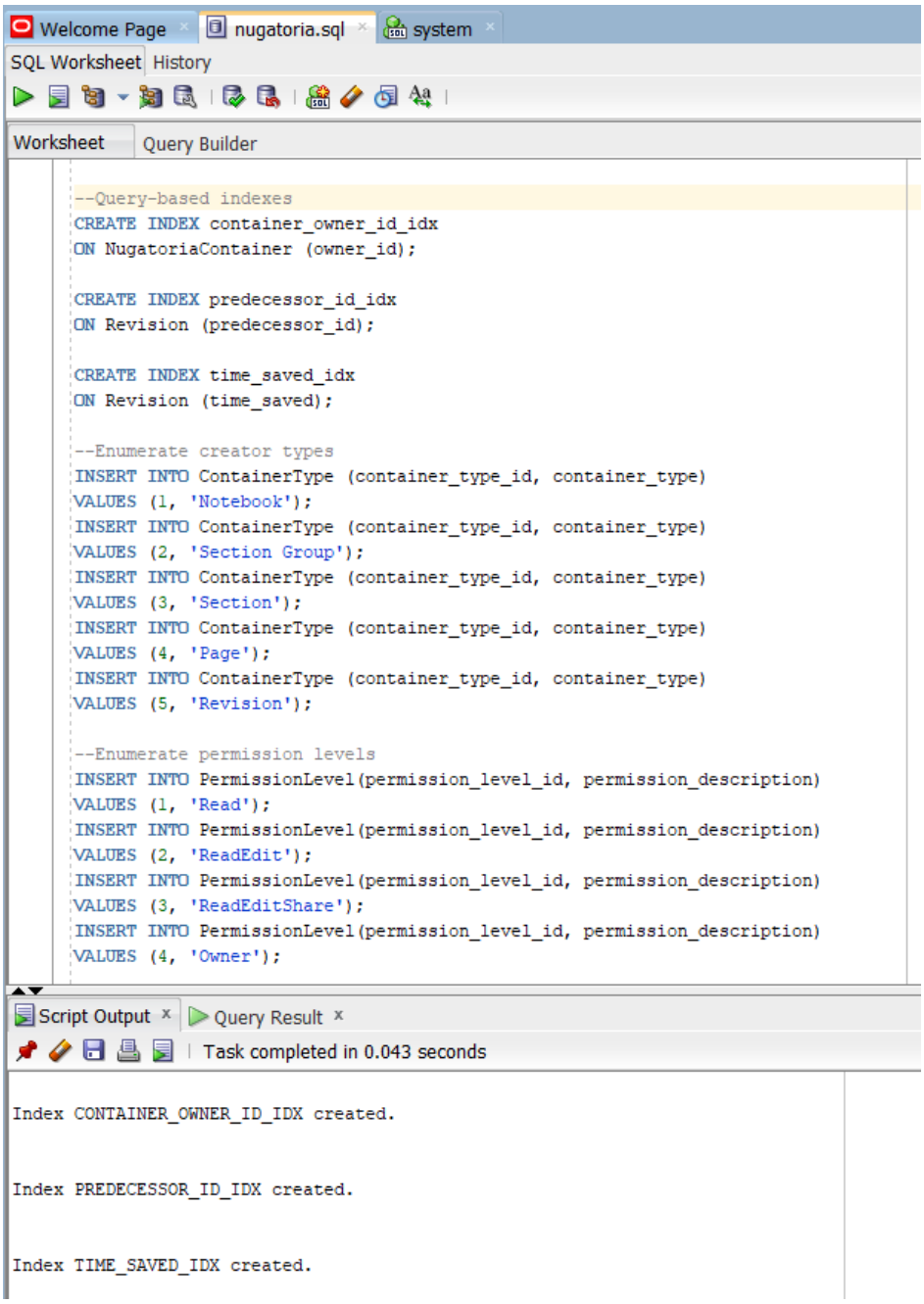
3. Finally, a common query would be to find all notebooks for which a given user has permission. For example, if we are looking for all notebooks for which a user with account_id=1 has permission, we could use the following query:

```
SELECT email, container_name AS notebook_name, permission_description

FROM Account

JOIN NugatoriaPermission ON NugatoriaPermission.account_id = Account.account_id

JOIN PermissionLevel ON NugatoriaPermission.permission_level_id =
PermissionLevel.permission_level_id

JOIN NugatoriaContainer ON NugatoriaPermission.container_id =
NugatoriaContainer.container_id

WHERE Account.account_id = 1;
```

The attributes used in the JOIN statements and WHERE statements are all foreign keys, and do not provide further ideas for indexing.

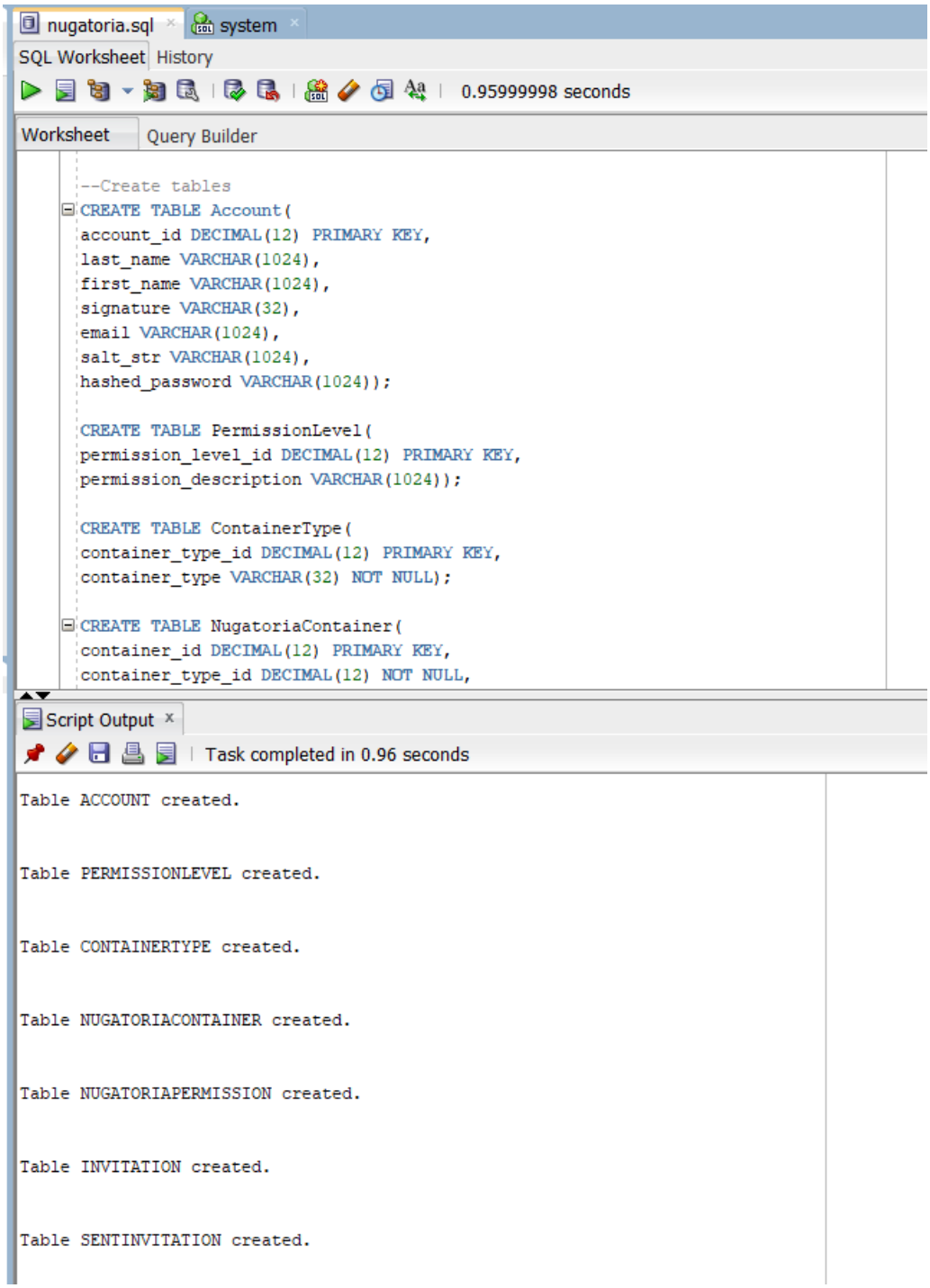
Below are screenshots of the index creation


nugatoria.sql
Welcome Page
system
SQL Worksheet
History
1.81099999 seconds
Worksheet
Query Builder
--Primary key indexes are created automatically
--Foreign key indexes:
CREATE INDEX invitation_permission_level_id_idx
ON Invitation (permission_level_id);
CREATE INDEX invitation_container_id_idx
ON Invitation (container_id);
CREATE INDEX permission_permissionlevel_id_idx
ON NugatoriaPermission (permission_level_id);
CREATE INDEX container_containertype_id_idx
ON NugatoriaContainer (container_type_id);
CREATE INDEX balance_change_account_id_idx
ON AccountBalanceChange (account_id);
CREATE INDEX owning_account_idx
ON Notebook (owning_account);
Script Output
Task completed in 1.811 seconds
Index INVITATION_PERMISSION_LEVEL_ID_IDX created.
Index INVITATION_CONTAINER_ID_IDX created.
Index PERMISSION_PERMISSIONLEVEL_ID_IDX created.
Index CONTAINER_CONTAINERTYPE_ID_IDX created.
Index BALANCE_CHANGE_ACCOUNT_ID_IDX created.
Index OWNING ACCOUNT IDX created.



Nugatoria Creation Script

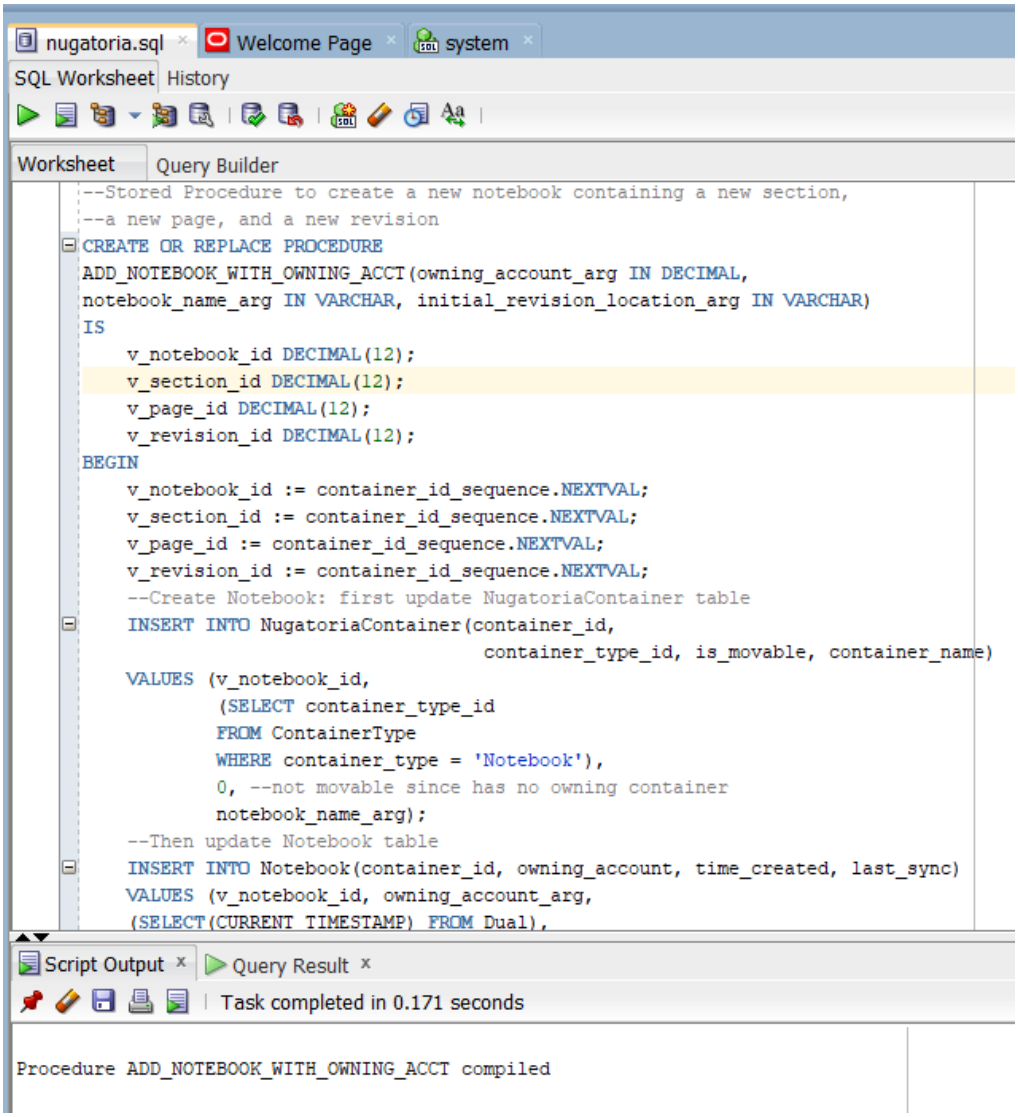
Below is a screenshot of a script that creates the Nugatoria database. The screenshot shows seven tables being successfully created, although the script does in fact create all the tables in the physical ERD.


nugatoria.sql
system
SQL Worksheet
History
0.95999998 seconds
Worksheet
Query Builder
--Create tables
CREATE TABLE Account(
account_id DECIMAL(12) PRIMARY KEY,
last_name VARCHAR(1024),
first_name VARCHAR(1024),
signature VARCHAR(32),
email VARCHAR(1024),
salt_str VARCHAR(1024),
hashed_password VARCHAR(1024));
CREATE TABLE PermissionLevel(
permission_level_id DECIMAL(12) PRIMARY KEY,
permission_description VARCHAR(1024));
CREATE TABLE ContainerType(
container_type_id DECIMAL(12) PRIMARY KEY,
container_type VARCHAR(32) NOT NULL);
CREATE TABLE NugatoriaContainer(
container_id DECIMAL(12) PRIMARY KEY,
container_type_id DECIMAL(12) NOT NULL,
Script Output
Task completed in 0.96 seconds
Table ACCOUNT created.
Table PERMISSIONLEVEL created.
Table CONTAINERTYPE created.
Table NUGATORIACONTAINER created.
Table NUGATORIAPERMISSION created.
Table INVITATION created.
Table SENTINVITATION created.

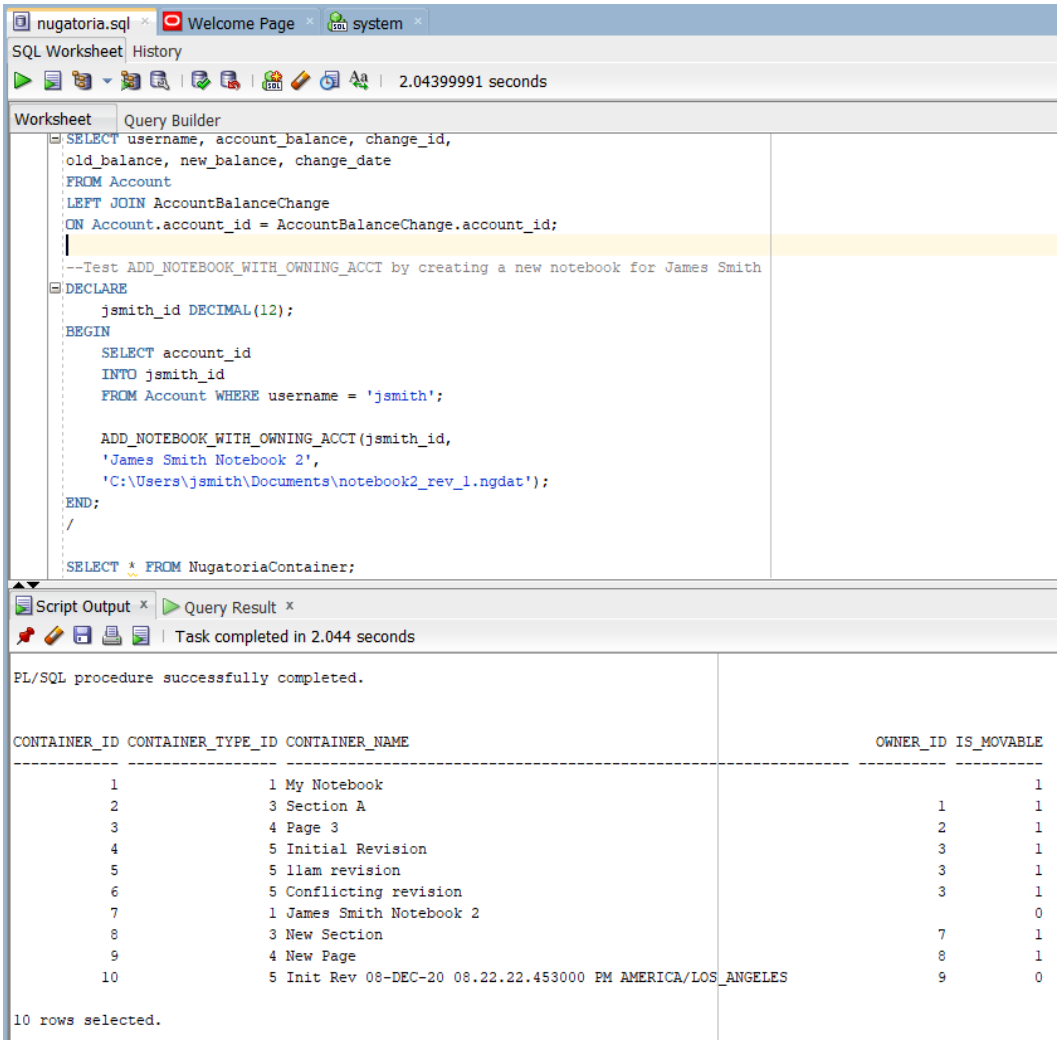
Reusable, Transaction-Oriented Stored Procedures

Stored Procedure to Create a New Notebook Associated with a Parameterized User Account

When a new user creates an account ("Use Case: User creates an account"), a notebook is automatically created for them. We will therefore have use for a stored procedure that creates a notebook, including a new section, a new page, and an initial revision of that first page. Below is a screenshot of the procedure compilation. Its full code is lengthy and may be found under "Appendix: PL/SQL code for ADD_NOTEBOOK_WITH_OWNING_ACCT" and in the SQL script included with this project.

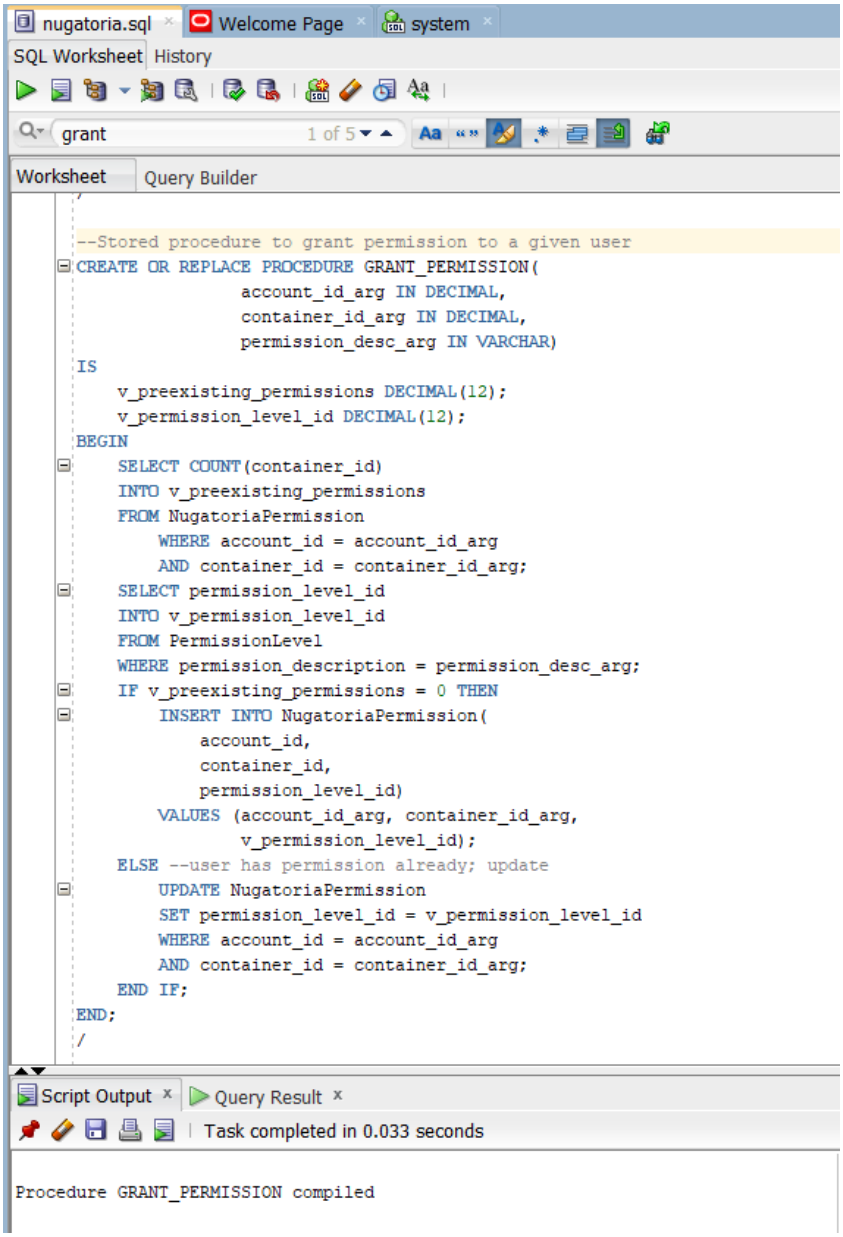


Because this procedure is important to the application, we should test it. Below we use the procedure to create a new notebook, named "James Smith Notebook 2", for the user James Smith. Note that the revision (`container_id = 10`) has owner "New Page" (`container_id = 9`), which in turn has owner "New Section" (`container_id = 8`), which in turn belongs to "James Smith Notebook 2" (`container_id = 7`).

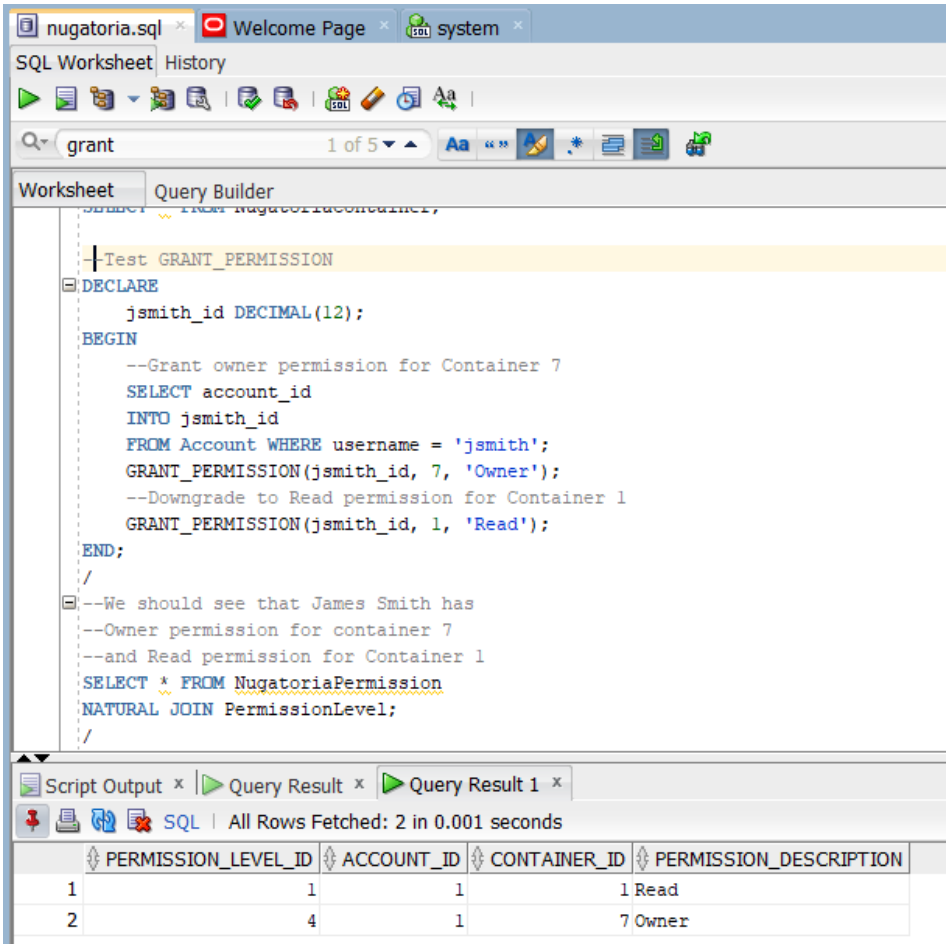

nugatoria.sql
Welcome Page
system
SQL Worksheet
History
2.04399991 seconds
Worksheet
Query Builder
SELECT username, account_balance, change_id,
old_balance, new_balance, change_date
FROM Account
LEFT JOIN AccountBalanceChange
ON Account.account_id = AccountBalanceChange.account_id;
--Test ADD_NOTEBOOK_WITH_OWNING_ACCT by creating a new notebook for James Smith
DECLARE
jsmith_id DECIMAL(12);
BEGIN
SELECT account_id
INTO jsmith_id
FROM Account WHERE username = 'jsmith';
ADD_NOTEBOOK_WITH_OWNING_ACCT(jsmith_id,
'James Smith Notebook 2',
'C:\Users\jsmith\Documents\notebook2_rev_1.ngdat');
END;
/
SELECT * FROM NugatoriaContainer;
Script Output
Query Result
Task completed in 2.044 seconds
PL/SQL procedure successfully completed.
CONTAINER_ID CONTAINER_TYPE_ID CONTAINER_NAME OWNER_ID IS_MOVABLE
1 1 My Notebook 1
2 3 Section A 1 1
3 4 Page 3 2 1
4 5 Initial Revision 3 1
5 5 llam revision 3 1
6 5 Conflicting revision 3 1
7 1 James Smith Notebook 2 0
8 3 New Section 7 1
9 4 New Page 8 1
10 5 Init Rev 08-DEC-20 08.22.22.453000 PM AMERICA/LOS_ANGELES 9 0
10 rows selected.

Stored Procedure to Grant Permission

At least two use cases (“Use Case: User creates an account” and “Use Case: User A shares notebook with User B”) involve updating the permission table to declare that a given user has permission to access a given notebook. We create a stored procedure to store that permission in the database. Below is a screenshot of the code and its successful compilation.

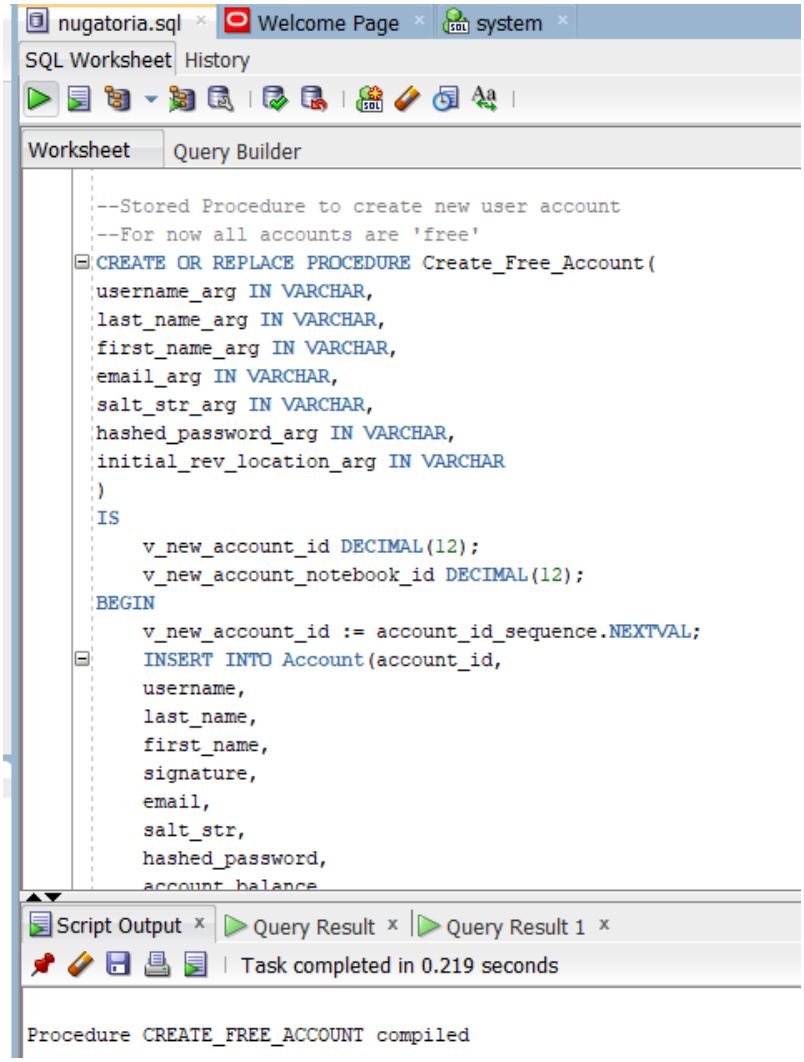


We test the code by using it to update two permissions for James Smith: he is granted Owner permission for Container 7, and his permission is downgraded from Owner to Read for Container 1. Screenshot below.

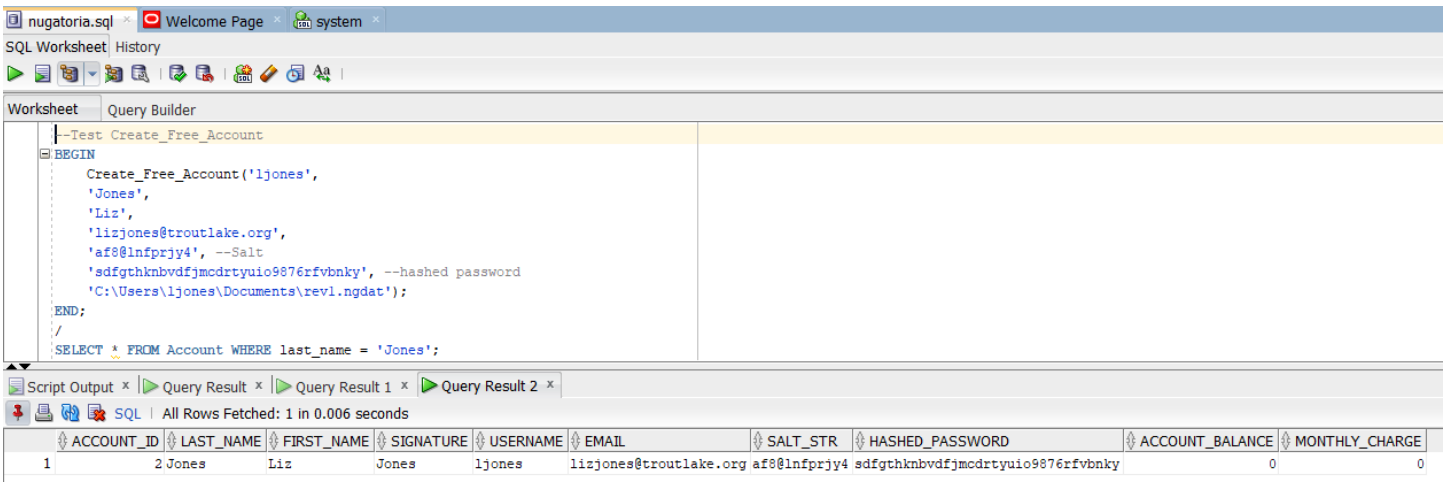


Stored Procedure to Create Account

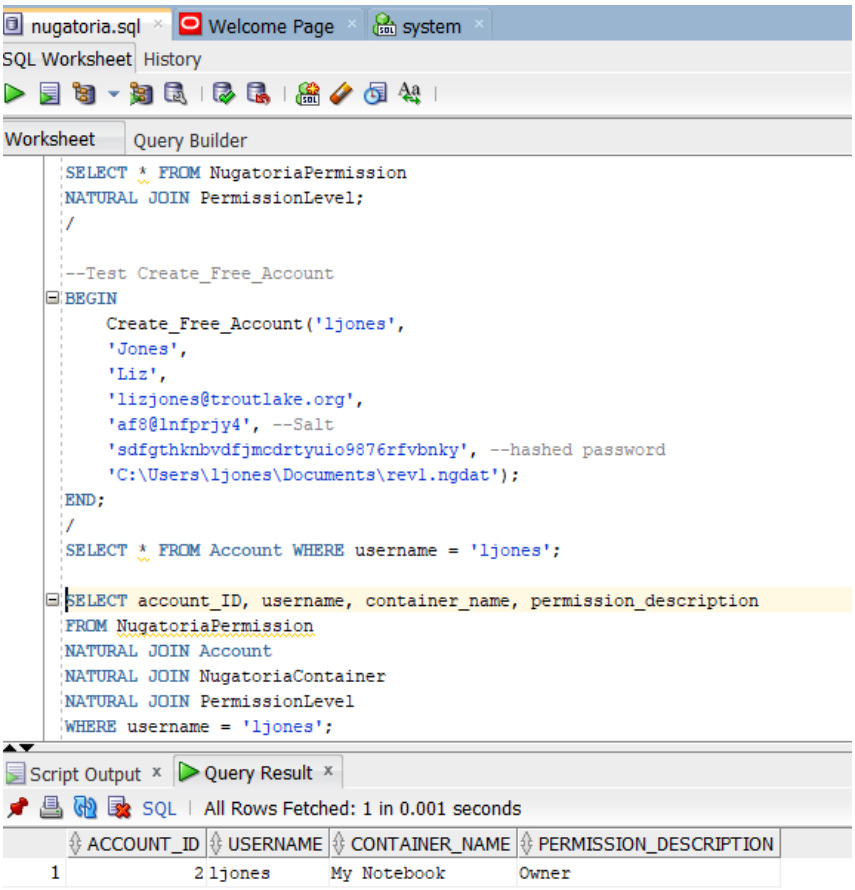
Thirdly, we create a stored procedure to create a new user account. This involves creating a new notebook and grating the new user Owner permission for that notebook. A screenshot of the procedure compilation is below. The entire code is in the attached SQL script and in "Appendix: PL/SQL Code for CREATE_FREE_ACCOUNT."



To test the procedure we create a new user, Liz Jones. The new account has been inserted into the Account table, as shown below.



Furthermore, there is a new notebook for which Liz Jones has Owner-level permission:



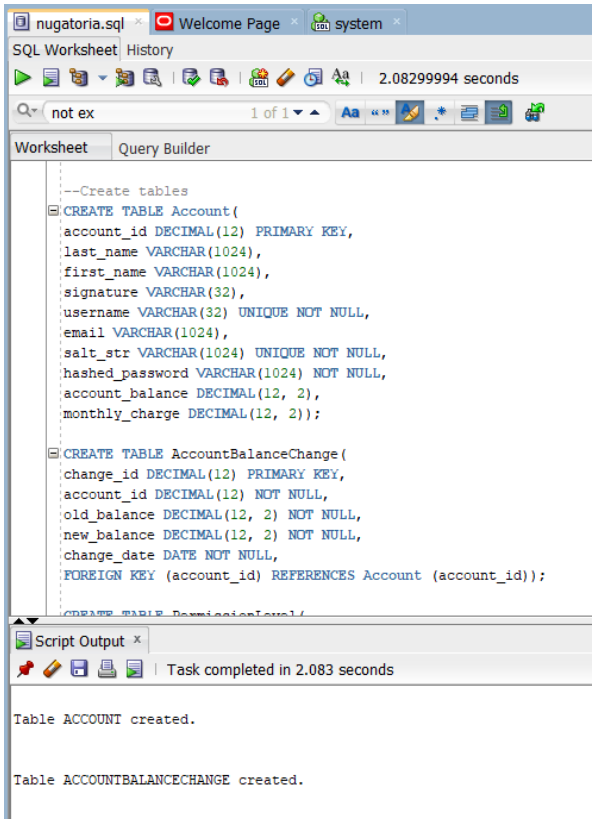
History Table

Tracking the history of Account Balances

Initially, Nugatoria will be a free application. However, it is possible that it might move to a subscription model in the future. It might be helpful to keep track of a user's account balance. For example, there might be a model where a user can choose to pay a monthly fee or pay an annual sum up front for a reduced rate; in the latter case, we could credit the annual sum to the user's account balance and then charge the reduced monthly rate against that sum. It will also be of interest to examine the history of a user's account balance. Therefore, I have added an AccountBalanceChange table to my database design to track changes over time. This table implies a new structural rule: Every account may be referenced by many AccountBalanceChanges; each AccountBalanceChange references one account. The attributes are described above (Table: AccountBalanceChange).

We create the Account table and AccountBalanceChange in SQL using the following code:

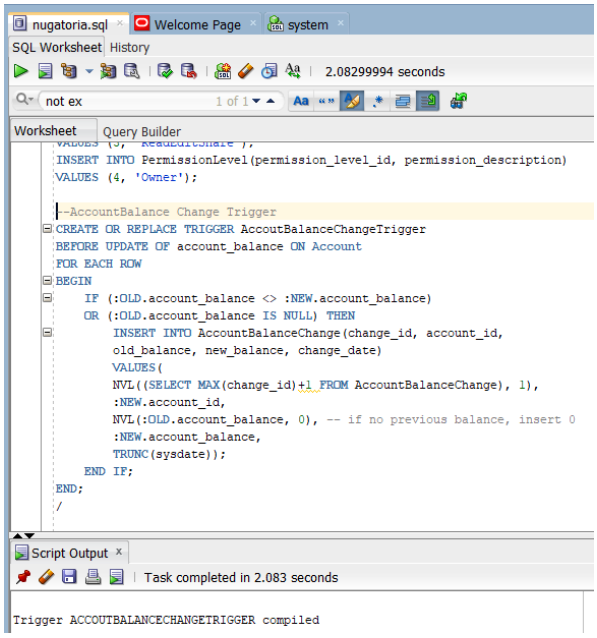
```
CREATE TABLE Account(
account_id DECIMAL(12) PRIMARY KEY,
last_name VARCHAR(1024),
first_name VARCHAR(1024),
signature VARCHAR(32),
username VARCHAR(32) UNIQUE NOT NULL,
email VARCHAR(1024),
salt_str VARCHAR(1024) UNIQUE NOT NULL,
hashed_password VARCHAR(1024) NOT NULL,
account_balance DECIMAL(12, 2),
monthly_charge DECIMAL(12, 2));

CREATE TABLE AccountBalanceChange(
change_id DECIMAL(12) PRIMARY KEY,
account_id DECIMAL(12) NOT NULL,
old_balance DECIMAL(12, 2) NOT NULL,
new_balance DECIMAL(12, 2) NOT NULL,
change_date DATE NOT NULL,
FOREIGN KEY (account_id) REFERENCES
Account (account_id));
```



Creating a Trigger for Account Balance Changes

In order to ensure that the AccountBalanceChanges table is up to date, we create a trigger to update the table every time an update is made to the account_balance attribute of the Acccount table. The code for this trigger is shown below, along with a screenshot showing its successful compilation:

```
CREATE OR REPLACE TRIGGER
AccoutBalanceChangeTrigger
BEFORE UPDATE OF account_balance ON Account
FOR EACH ROW
BEGIN
    IF (:OLD.account_balance <>
:NEW.account_balance)
    OR (:OLD.account_balance IS NULL) THEN
        INSERT INTO
AccountBalanceChange(change_id, account_id,
        old_balance, new_balance, change_date)
        VALUES(
        NVL((SELECT MAX(change_id)+1 FROM
AccountBalanceChange), 1),
        :NEW.account_id,
        NVL(:OLD.account_balance, 0),
        :NEW.account_balance,
        TRUNC(sysdate));
    END IF;
END;
/
```



Let's step through this code piece by piece:

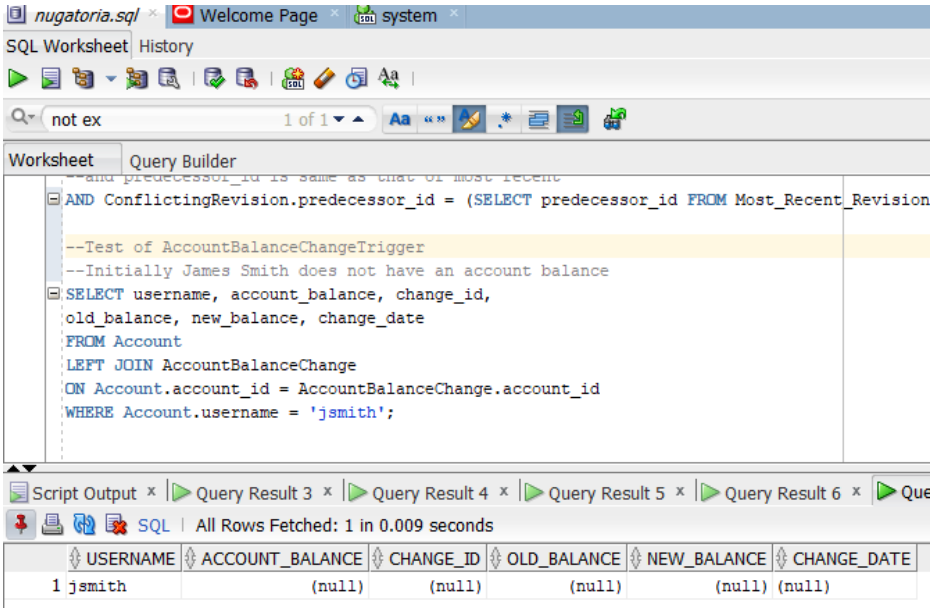
`CREATE OR REPLACE TRIGGER AccoutBalanceChangeTrigger BEFORE UPDATE OF account_balance ON Account`	Define a trigger that will execute before the account_balance attribute of the Account table is updated.
`FOR EACH ROW`	Declare this trigger to be a row-level trigger so that it will execute once for each affected row.
`BEGIN`	Begin the trigger code block

`IF (:OLD.account_balance <> :NEW.account_balance)`
`OR (:OLD.account_balance IS NULL)`
`THEN`
	Only update the AccountBalanceChange table if the account balance has changed from its previous value or if the previous account balance was NULL

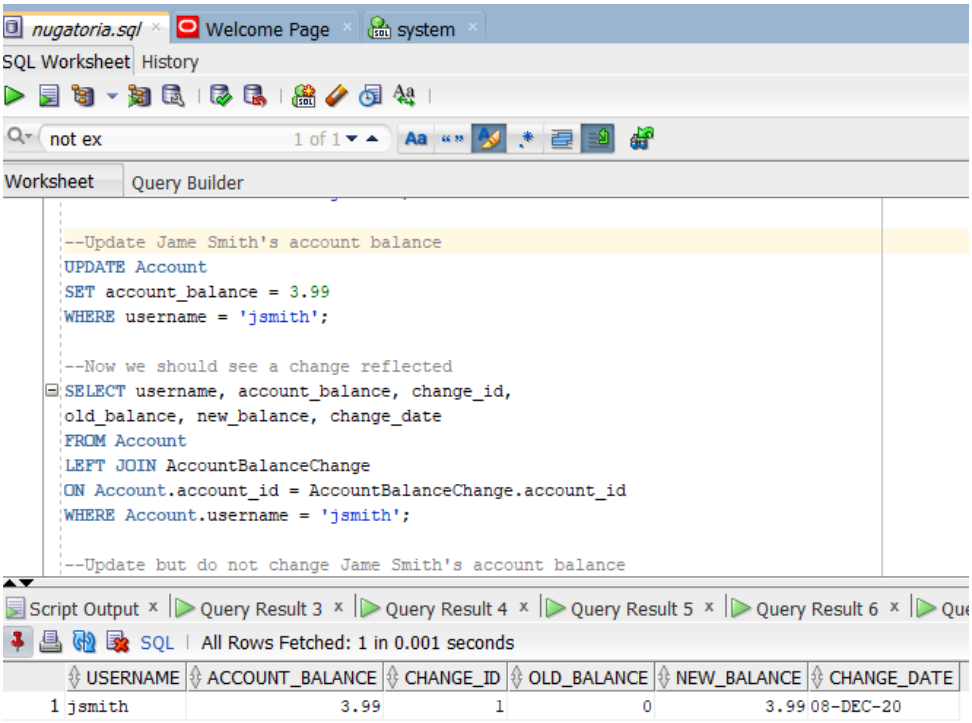
`INSERT INTO AccountBalanceChange(change_id, account_id,` `old_balance, new_balance, change_date`	Create a new record on the AccountBalanceChange table and specify values for all the attributes
`VALUES(` `NVL((SELECT MAX(change_id)+1 FROM AccountBalanceChange), 1),`	The change_id of the current change is equal to the largest current primary key value plus one (hence it is unique and nonnull). If there are no values in the table, insert 1 as the new key.
`:NEW.account_id,`	Fetch the account_id from the :NEW pseudotable
`NVL(:OLD.account_balance, 0),`	Fetch the old account balance or insert 0 if there was no data found
`:NEW.account_balance,`	Fetch the new account balance
`TRUNC(sysdate));`	Insert the current day
`END IF;`	End the IF block
`END;`	End the trigger block
`/`	Execute the buffer to create the trigger

We illustrate this trigger's execution by creating a user, James Smith, who initially has no (i.e. NULL) account balance (Step 1). His account balance is updated to $3.99, and the trigger records the change in the AccountBalanceChange table (Step 2). His account balance is again updated to $3.99, and since there is no change, the trigger does not update the AccountBalanceChange table (Step 3). Finally (Step 4), we change the account balance again and see that the trigger has updated the AccountBalanceChange table to contain two records.

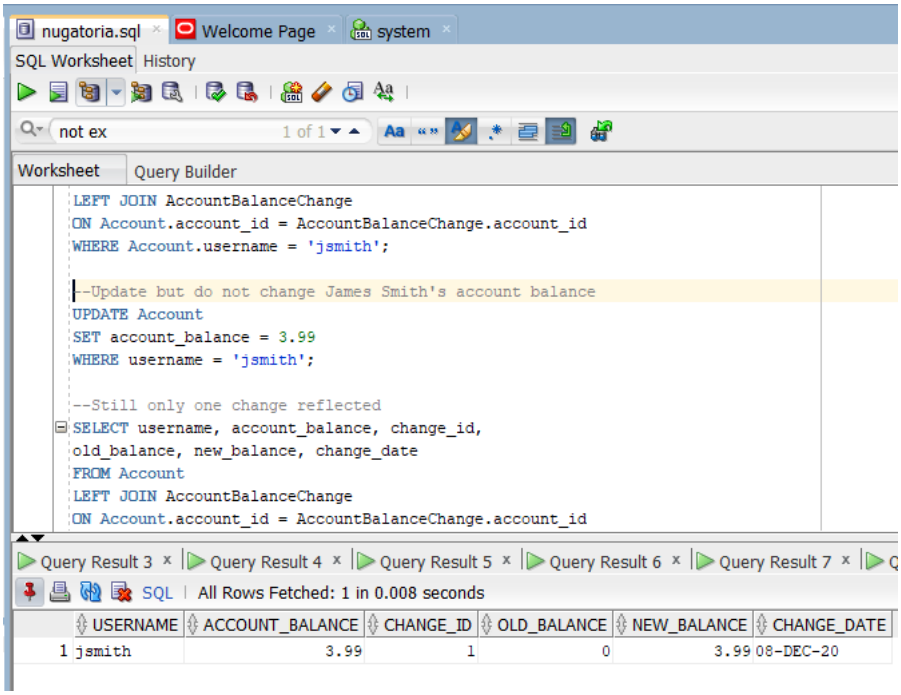
Step 1: Initially James Smith has NULL account balance



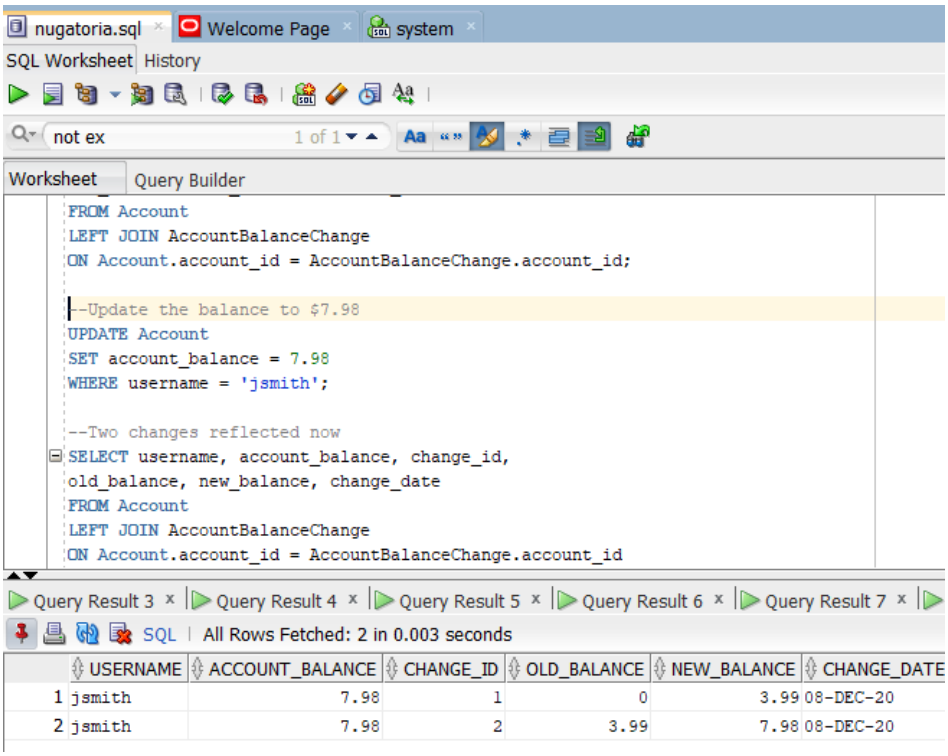
Step 2: Update James Smith's account balance to 3.99; the trigger executes



Step 3: Update but do not change James Smith's balance. The trigger does not record a change. We see only the change that was recorded in Step 2 (`change_id = 1`).



Step 4: Change the account balance again; we now see that two changes have been recorded.



Questions and Queries

Query: Identifying Edit Conflicts

One common query for the app would be to identify edit conflicts. An edit conflict occurs when two revisions of a given page have the same predecessor. The only way for that circumstance to occur is if a user tries to edit an out-of-date revision.

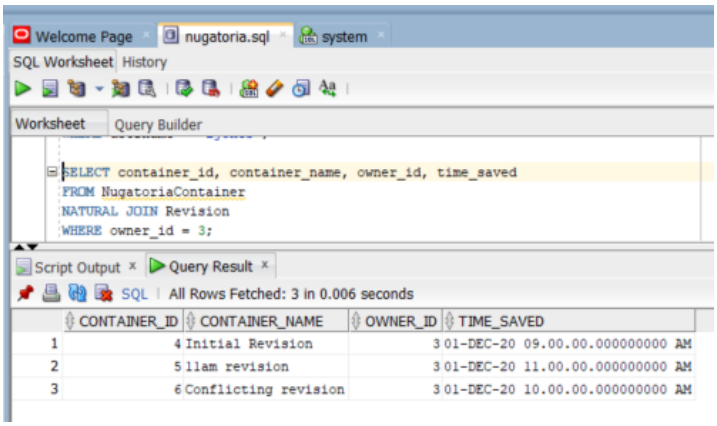
For example, suppose Alice creates a page at 9am (`container_id = 4`).

At 10am she edits the page on her phone, creating a new revision. Her phone has poor internet and does not store the page in the database, so that revision does not yet have a container id, though it does have a `predecessor_id` of 4, the `container_id` of the preceding revision.

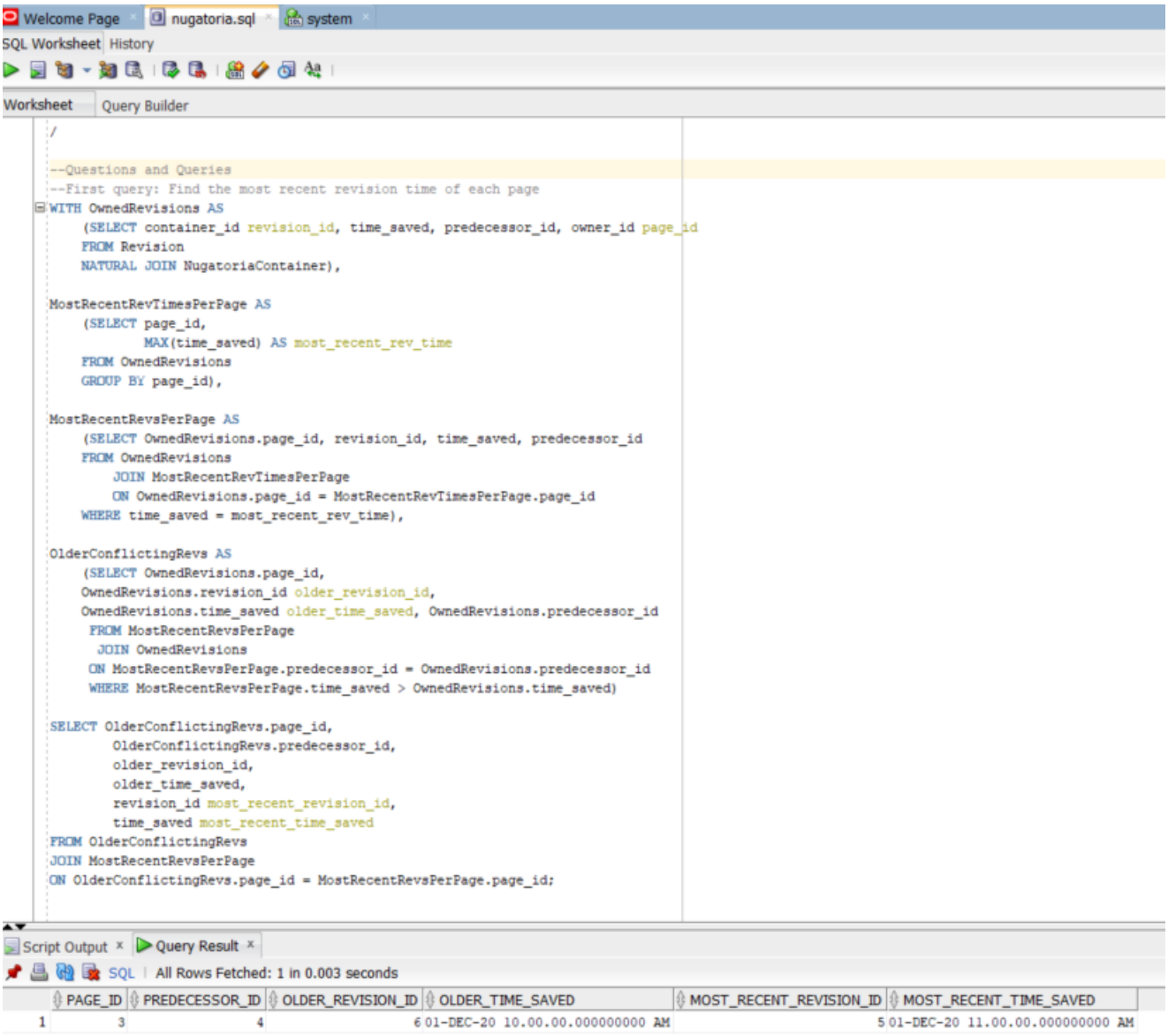
At 11am she edits the page on her computer, thereby creating another revision, again with `predecessor_id = 4`. This 11am revision is stored in the database successfully and has `container_id = 5`.

Her phone finally syncs, so her 10am revision is saved in the database with `container_id = 6`.

Since the 10am revision and the 11am revision are both preceded by the revision with `container_id=4`, we have an edit conflict. These three revisions are shown in the screenshot below.



The following query identifies all edit conflicts relevant to the most recent revision of each page. As expected, it flags the edit conflict described above. The query meets Group 1 requirements by restricting rows with a WHERE clause and it meets Group 2 requirements by using the aggregate function MAX, as well as multiple (named) subqueries.



To understand the logic of this query, we break it into smaller sections:

`WITH OwnedRevisions AS`
`(SELECT container_id revision_id, time_saved, predecessor_id, owner_id page_id`
`FROM Revision`
`NATURAL JOIN NugatoriaContainer),`
	We are interested in looking at different revisions of the same page, with the same predecessor, and at their times. The fact that a revision belongs to a page is represented by the owner_id attribute of the NugatoriaContainer table; the other attributes are on the Revision table. We therefore join these two tables on their primary key (container_id) and select the relevant data in a named subquery. For convenience and clarity, we also use aliases: container_id is renamed to revision_id (since we are interested in the id numbers of revisions specifically), while owner_id is renamed to page_id (since revisions are owned by pages).

`MostRecentRevTimesPerPage AS`
`(SELECT page_id,`
`MAX(time_saved) AS most_recent_rev_time`
	This subquery identifies the most recent time_saved among all revisions, grouped by page_id. It returns a set of (page_id, most_recent_time_saved) pairs.

`FROM OwnedRevisions` `GROUP BY page_id),`	

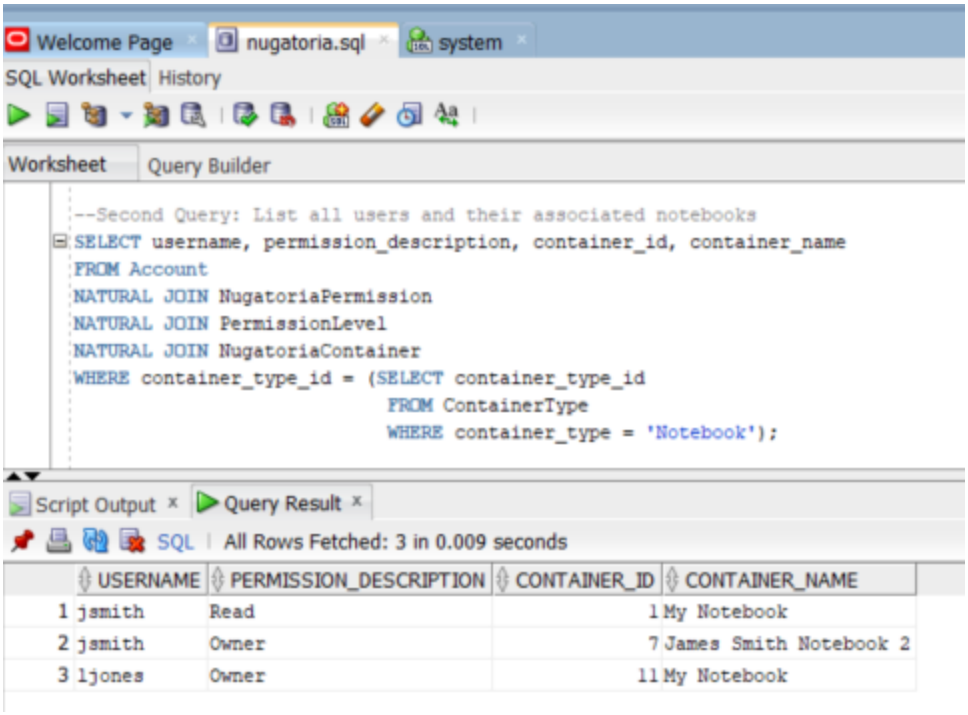
`MostRecentRevsPerPage AS`
`(SELECT OwnedRevisions.page_id, revision_id, time_saved, predecessor_id`
`FROM OwnedRevisions`
`JOIN MostRecentRevTimesPerPage`
`ON OwnedRevisions.page_id = MostRecentRevTimesPerPage.page_id`
`WHERE time_saved = most_recent_rev_time),`
	Since the subquery MostRecentRevTimesPerPage does not include any data about the most recent revision other than its save time, we join MostRecentRevTimesPerPage with OwnedRevision on their respective page_id columns. This allows us to select the revision_id and predecessor_id of the revision with the most recent time_saved value.

`OlderConflictingRevs AS`
`(SELECT OwnedRevisions.page_id,`
`OwnedRevisions.revision_id older_revision_id,`
`OwnedRevisions.time_saved older_time_saved,`
`OwnedRevisions.predecessor_id`
`FROM MostRecentRevsPerPage`
`JOIN OwnedRevisions`
`ON MostRecentRevsPerPage.predecessor_id = OwnedRevisions.predecessor_id`
`WHERE MostRecentRevsPerPage.time_saved > OwnedRevisions.time_saved)`
	This subquery joins the MostRecentRevsPerPage subquery with OwnedRevisions on their shared predecessor_id; that is, it finds all revisions that share a predecessor with the most recent revision of each page. The WHERE statement restricts the result set to only those revisions that are older than the most recent revision, i.e. those revisions involved in edit conflicts.

`SELECT OlderConflictingRevs.page_id,`
 	
`OlderConflictingRevs.predecessor_id,`
`older_revision_id,`
`older_time_saved,`
`revision_id most_recent_revision_id,`
`time_saved most_recent_time_saved`
`FROM OlderConflictingRevs`
`JOIN MostRecentRevsPerPage`
`ON OlderConflictingRevs.page_id = MostRecentRevsPerPage.page_id;`
	Finally, we extract useful and human-readable information about the two conflicting revisions by joining the MostRecentRevsPerPage and OlderConflictingRevs subqueries and selecting key data about the revisions in each conflict.

Query: Listing Users and Notebooks

A second question that the application will need to answer is *Which users have access to which notebooks*? This question would be useful for analyzing statistics, such as how many notebooks a user has access to on average. It would also be useful to know what kind of permission each user has. The following query returns the desired result. It meets the Group 1 requirements by joining at least two tables and it meets the Group 2 requirements by using a subquery. As illustrated in the screenshot below, it successfully identifies that user ‘jsmith’ has access to two notebooks (one with Read permission, one with Owner permission) and that user ‘ljones’ has Owner permission for her single notebook.



The logic of this query is fairly straightforward. We join the Account table with NugatoriaPermission using a natural join so that the two tables are joined on the account_id column. We join the resulting table with PermissionLevel using a natural join so that the two tables are joined on the permission_level_id column. We join that table with NugatoriaContainer using a natural join so that the two tables are joined on the container_id column. The select statement from that unified table would select the username, permission description, container_id, and container_name for all users with permission to all accounts. We restrict the results using a WHERE clause and a subquery in order to see only notebooks.

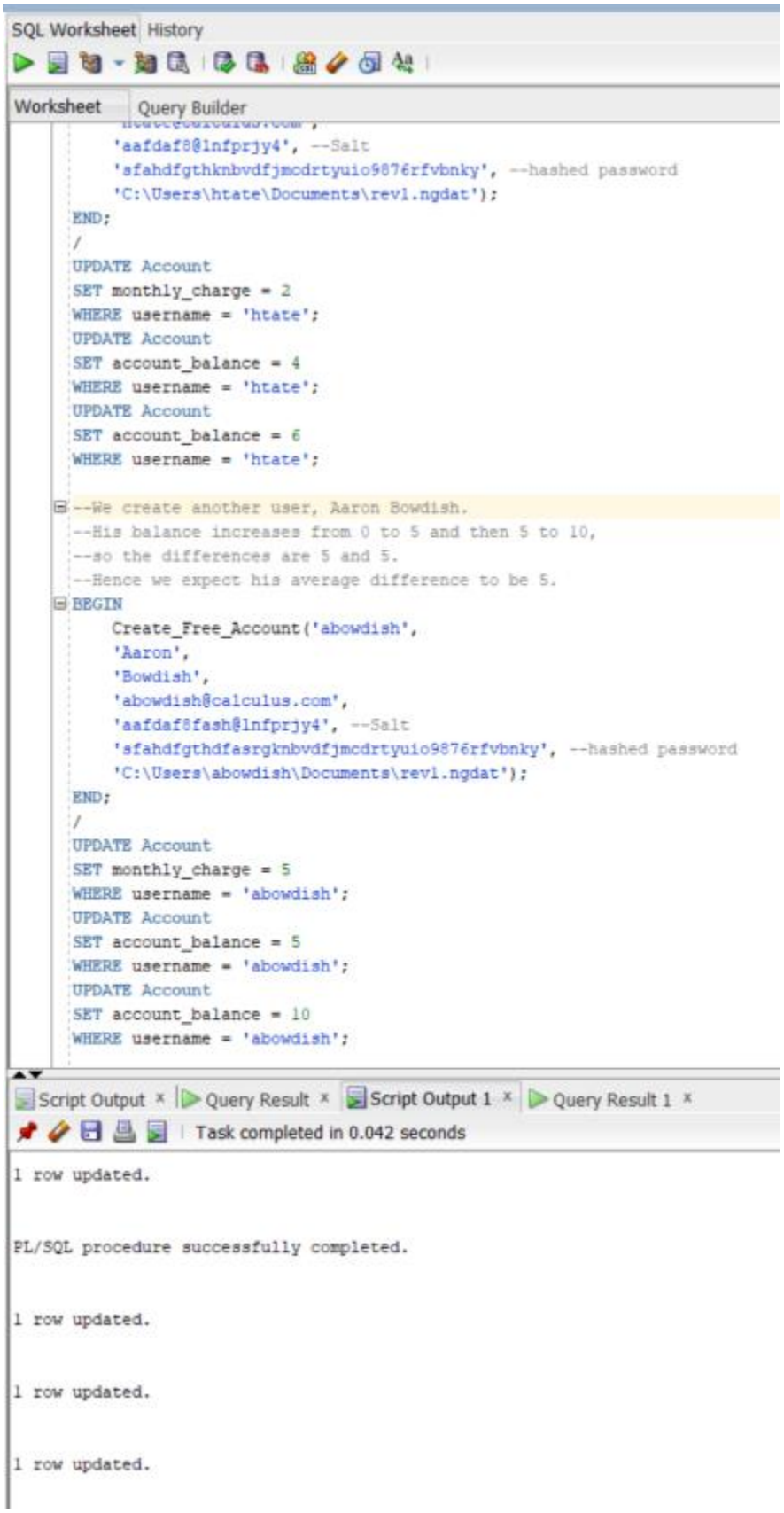
Query: Average Monthly Balance Per User Over Previous Year

A third useful query would be to examine the average balance of customers over time. Since some of our customers will have free accounts, we are not particularly interested in their balances, so we look only at balances for users who have a positive monthly charge.

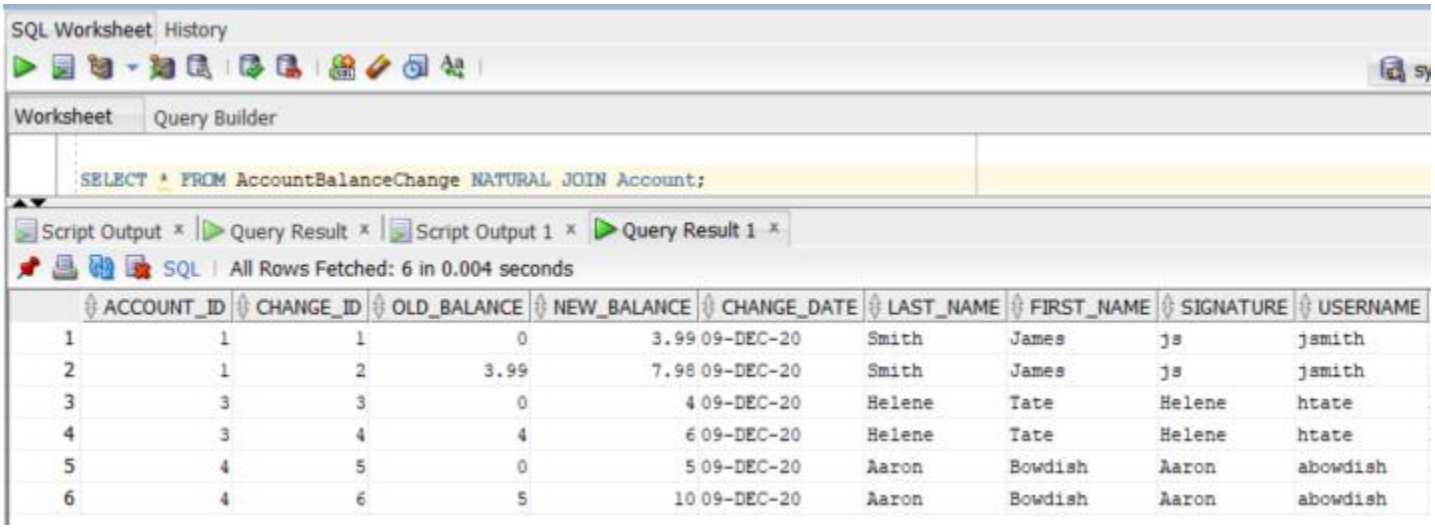
First, we create two paid users and create a balance history. Our first user, Helene Tate, has a monthly charge of $2.00. Her balance is initially $0.00, then rises to $4.00 and then to $6.00. Since the differences are $4.00 and $2.00, the average change of her balance over time is $3.00. Below is a screenshot of her data being entered into the database.


SQL Worksheet
History
Worksheet
Query Builder
NATURAL JOIN Revision
WHERE owner_id = 3;
--Insert data for testing query about average change in balance per use
--We create a user, Helene Tate.
--Her balance increases from 0 to 4 and then 4 to 6,
--so the differences are 4 and 2.
--Hence we expect her average difference to be 3.
BEGIN
Create_Free_Account('htate',
'Helene',
'Tate',
'htate@calculus.com',
'aafdaf8@lnfprjy4', --Salt
'sfahdfgthknbvdfjmcdrtyuio9876rfvbnky', --hashed password
'C:\Users\htate\Documents\revl.ngdat');
END;
/
UPDATE Account
SET monthly_charge = 2
WHERE username = 'htate';
UPDATE Account
SET account_balance = 4
WHERE username = 'htate';
UPDATE Account
SET account_balance = 6
WHERE username = 'htate';
--We create another user, Aaron Bowdish.
--His balance increases from 0 to 5 and then 5 to 10,
--so the differences are 5 and 5.
--Hence we expect his average difference to be 5.
BEGIN
Create_Free_Account('abowdish',
'Aaron',
'Bowdish',
'abowdish@calculus.com',
'aafdaf8fash@lnfprjy4', --Salt
'sfahdfgthdfasrgknbvdfjmcdrtyuio9876rfvbnky', --hashed password
'C:\Users\abowdish\Documents\revl.ngdat');
Script Output
Query Result
Script Output 1
Query Result 1
Task completed in 0.062 seconds
PL/SQL procedure successfully completed.
1 row updated.
1 row updated.
1 row updated.

We next create a user, Aaron Bowdish, whose balance increases from $0.00 to $5.00 and then to $10.00, so his average balance change is $5.00. Below is a screenshot of his data being entered.

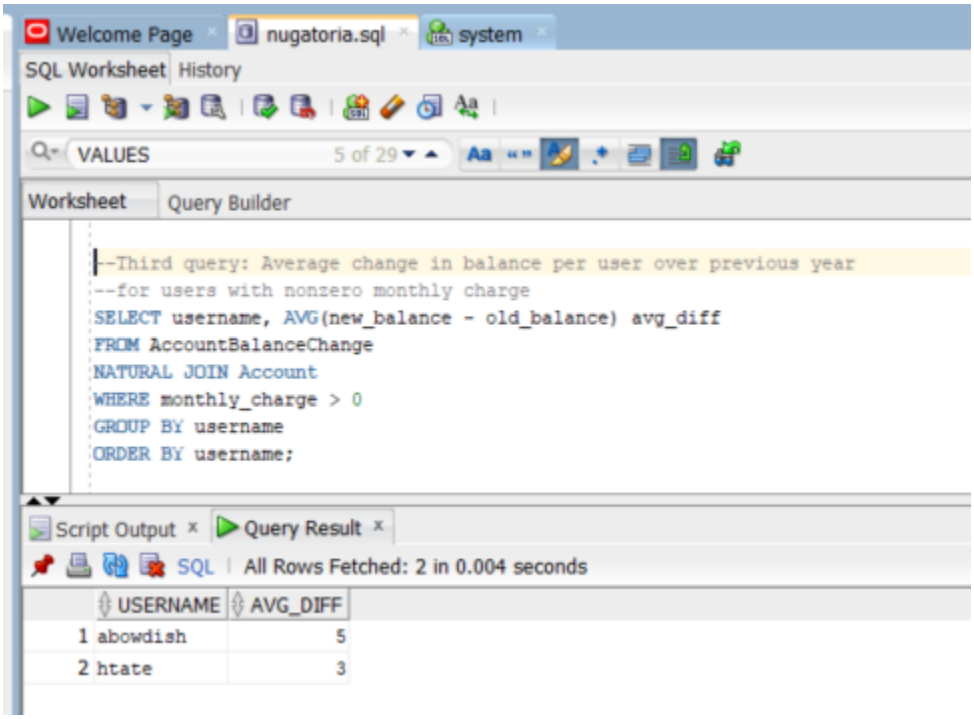


We verify that those two users (among others) are present in our Account table with their changes recorded in the AccountBalanceChange table.



	ACCOUNT_ID	CHANGE_ID	OLD_BALANCE	NEW_BALANCE	CHANGE_DATE	LAST_NAME	FIRST_NAME	SIGNATURE	USERNAME
1	1	1	0	3.99	09-DEC-20	Smith	James	js	jsmith
2	1	2	3.99	7.98	09-DEC-20	Smith	James	js	jsmith
3	3	3	0	4	09-DEC-20	Helene	Tate	Helene	htate
4	3	4	4	6	09-DEC-20	Helene	Tate	Helene	htate
5	4	5	0	5	09-DEC-20	Aaron	Bowdish	Aaron	abowdish
6	4	6	5	10	09-DEC-20	Aaron	Bowdish	Aaron	abowdish

To find the average balance change of each account, we execute the following query. Note that the avg_diff column does give the values we expected ($5.00 for Aaron Bowdish and $3.00 for Helene Tate). Note also that, although James Smith is present in the AccountBalanceChange table (above), he is not included in the result of the query since he has no monthly charge.



	USERNAME	AVG_DIFF
1	abowdish	5
2	htate	3

This query selects two columns, the username, and the average change in account balance. We use a GROUP BY clause to group the results by username so that the query returns the average account balance change for each individual user. We also use an ORDER BY clause to display the results alphabetically by username.

This query meets the Group 1 requirements by using a WHERE clause to restrict the results, and it meets the Group 2 requirements by using an aggregate function.

Summary and Reflection

This project is very much a beginning, not an endpoint. Nugatoria is an application that will require further design and development in order to be meaningfully used, and much of that development will happen in a language such as Java or Python. However, this project has established a database as a solid foundation for the application. The design is laid out in entity-relationship diagrams. I used the entities to design tables for a relational database model. The tables have been normalized. I have improved efficiency by adding indexes. I have developed queries and stored procedures to perform a few common and useful tasks. The work here provides strong material to build on in the future.

References

Coronel, C., & Morris, S. (2018). *Database Systems: Design, Implementation, & Management.* Cengage Learning, Inc.

Lewis, C. T., & Short, C. (1870). *A Latin Dictionary.* New York: Harper and Brothers. Retrieved from https://www.latinitium.com/latin-dictionaries?t=lsn31375

Appendix: PL/SQL code for ADD_NOTEBOOK_WITH_OWNING_ACCT

```
--Stored Procedure to create a new notebook containing a new section,
--a new page, and a new revision
CREATE OR REPLACE PROCEDURE
ADD_NOTEBOOK_WITH_OWNING_ACCT(owning_account_arg IN DECIMAL,
notebook_name_arg IN VARCHAR, initial_revision_location_arg IN VARCHAR)
IS
    v_notebook_id DECIMAL(12);
    v_section_id DECIMAL(12);
    v_page_id DECIMAL(12);
    v_revision_id DECIMAL(12);
BEGIN
    v_notebook_id := container_id_sequence.NEXTVAL;
    v_section_id := container_id_sequence.NEXTVAL;
    v_page_id := container_id_sequence.NEXTVAL;
    v_revision_id := container_id_sequence.NEXTVAL;
    --Create Notebook: first update NugatoriaContainer table
    INSERT INTO NugatoriaContainer(container_id,
                                   container_type_id, is_movable, container_name)
    VALUES (v_notebook_id,
            (SELECT container_type_id
            FROM ContainerType
            WHERE container_type = 'Notebook'),
            0, --not movable since has no owning container
            notebook_name_arg);
    --Then update Notebook table
    INSERT INTO Notebook(container_id, owning_account, time_created, last_sync)
    VALUES (v_notebook_id, owning_account_arg,
```

```
    (SELECT(CURRENT_TIMESTAMP) FROM Dual),
    (SELECT(CURRENT_TIMESTAMP) FROM Dual));
    --Create section
  INSERT INTO NugatoriaContainer(container_id,
                                 container_type_id, owner_id,
                                 is_movable, container_name)
    VALUES (v_section_id,
           (SELECT container_type_id FROM ContainerType
            WHERE container_type = 'Section'),
            v_notebook_id, --owned by notebook
            1, --sections are movable
            'New Section');
    --Create page
    INSERT INTO NugatoriaContainer(container_id,
                                 container_type_id, owner_id,
                                 is_movable, container_name)
    VALUES (v_page_id,
           (SELECT container_type_id FROM ContainerType
            WHERE container_type = 'Page'),
            v_section_id, --owned by section
            1, --pages are movable
            'New Page');
    --Create revision: update NugatoriaContainer table
    INSERT INTO NugatoriaContainer(container_id,
                                 container_type_id, owner_id,
                                 is_movable, container_name)
    VALUES (v_revision_id,
           (SELECT container_type_id FROM ContainerType
            WHERE container_type = 'Revision'),
            v_page_id, --owned by page
            0, --revisions are not movable
            'Init Rev ' ||
            TO_CHAR((SELECT(CURRENT_TIMESTAMP) FROM Dual)));
    --Update Revision table
    INSERT INTO Revision(container_id, location, time_saved, deprecated)
    VALUES (v_revision_id, initial_revision_location_arg,
    (SELECT(CURRENT_TIMESTAMP) FROM Dual),
    0);
END;
/
```

Appendix: PL/SQL Code for CREATE_FREE_ACCOUNT

```
--Stored Procedure to create new user account
--For now all accounts are 'free'
CREATE OR REPLACE PROCEDURE Create_Free_Account(
```

```
username_arg IN VARCHAR,
last_name_arg IN VARCHAR,
first_name_arg IN VARCHAR,
email_arg IN VARCHAR,
salt_str_arg IN VARCHAR,
hashed_password_arg IN VARCHAR,
initial_rev_location_arg IN VARCHAR
)
IS
    v_new_account_id DECIMAL(12);
    v_new_account_notebook_id DECIMAL(12);
BEGIN
    v_new_account_id := account_id_sequence.NEXTVAL;
    INSERT INTO Account(account_id,
    username,
    last_name,
    first_name,
    signature,
    email,
    salt_str,
    hashed_password,
    account_balance,
    monthly_charge)
    VALUES (v_new_account_id,
    username_arg, last_name_arg, first_name_arg,
    last_name_arg, -- by default, signature is last name
    email_arg,
    salt_str_arg,
    hashed_password_arg,
    0,0);
    ADD_NOTEBOOK_WITH_OWNING_ACCT(v_new_account_id,
    'My Notebook', initial_rev_location_arg);
    --Find ID of the (unique) notebook with this owning account
    --(unique because the account is new!)
    SELECT container_id
    INTO v_new_account_notebook_id
    FROM Notebook
    WHERE owning_account = v_new_account_id;
    GRANT_PERMISSION(v_new_account_id, v_new_account_notebook_id, 'Owner');
END;
/
```