Database Application Development

TeAmWORK submission

David Spokes, Mark Tomlin, Kudzai Maziva, Tinashe Jaravaza

[Year]

PHYSICAL DATABASE DESIGN PRO-FORMA

Contents

Data-types

Integrity Rules

Indexes

Sequences

Stored program Units

Datatypes

*This section should list both generalised data-types choices (eg all primary key columns to declared as 6 digit numbers), and more specific data-type choices and rationale for the data-types used for columns in the* ***new tables*** *added in Task 1.*

All primary key attributes, whether single or compound, are of integer data type. An arbitrary maximum of 4 digits, ie accommodating a maximum value of 9999, can be specified for these attributes.

- The description and name attributes **Email** etc are of text data type and should be set to variable string size of a maximum of 50

- Actual hours and estimated hours are of real data type with 1 decimal place, and a maximum value of 999.9

- Quantity attributes are integer to a maximum of 4 digits, ie 9999

- Fees basis should be CHAR(2) datatype

- Chargeable attributes are logical (see also integrity section)

- Unit cost is of real data type with 2 decimal places.

-Tel and fax nos are Integer numbers of 15 digits

-postcode is of varChar with a maximum length of 7 characters

-invoice date , due date , paid date are of type DATE datatype

Integrity Rules

*This section should give both generalised statements in relation to general strategies adopted (eg (i) primary and foreign keys will be implemented as per the ERD, (ii)all foreign key attributes will be declared NOT NULL except …..), and details a rationale for any specific additional constraints you consider appropriate. This section should cover* ***all tables (existing and new ones****). You may like to use the following format:*

|  |  |  |
| --- | --- | --- |
| Table name | Rule | Rationale |
| All | All foreign keys declared as not null except where indicated to the contrary on the ERD. |  |
| Charge\_grade | Hrly\_charge\_rate > 0 | To ensure that it does not cost the company money to do work. |
| Client | Email address must include the @ symbol and a dot after some text after the symbol  Telephone number and Fax number should only be numbers  PostCode should only contain letters and numbers (no symbols) | All valid email addresses include this symbol. |
| Project | Project\_code should be unique | This is a frequently used search criteria, duplicate project\_codes would cause confusion. |
| Stage | Fees basis can only take the values: ‘FQ’, ‘VS’, ‘VN’, ‘CS’, ‘CN’ | Consistent with business practice |
| Sessions and expense | Chargeable can only take the values Y or N and can not be null | Implements the Boolean value. It is essential that values are provided here to ensure that a range of reports including invoices are produced correctly. |
| Sessions | Actual\_hours > 0 and not null  Invoice\_no is initially NULL | Inorder to have a session , time must have been spent  invoice are only done after work has been completed (weekly , monthly) |
| invoice | Invoice\_date is not null and  Due date > invoice\_date  Paid date>invoice\_Date | Legal requirement.  Prevent potentially embarrassing data input errors being reflected on invoice documents. |
|  |  |  |

Indexes

*This section should give both generalised comments in relation to indexing strategy (eg all primary keys are automatically given a unique index, ) and specific index requirements* ***for all tables****.*

|  |  |  |
| --- | --- | --- |
| Table name | Column(s) unique/non-unique |  |
| All | Primary keys – Unique indexes | This is default Oracle processing |
| All | Foreign keys – non-unique indexes | Enhances performance of joins |
| whatever | Column X – non-unique index | This value used frequently as search criteria |

Sequences

*This section should list the sequences required (i) to generate* ***all*** *primary key values, (ii) for other reasons (if any).*

|  |  |  |  |
| --- | --- | --- | --- |
| Column Name | Table | Sequence Name | Starting value / Increment value |
| grade\_no | Charge-grade | g\_no\_in\_cg\_seq | 1/1 |
| task\_type\_no | Task\_type | tt\_no\_in\_tt\_seq | 10/10 |
| employee\_no | Employee | emp\_no\_in\_emp\_seq | 1/1 |
| project\_no | Project | p\_no\_in\_p\_seq | 1/1 |
| project\_code | Project | p\_c\_in\_p\_seq | 1/1 |
| stage\_id | Stage | s\_id\_in\_s\_seq | 1/1 |
| stage\_no | Stage | s\_no\_in\_s\_seq | 1/1 |
| task\_id | Task | t\_id\_in\_task\_seq | 1/1 |
| task\_no | Task | t\_no\_in\_t\_seq | 1/1 |
| assignment\_no | Assignment | a\_no\_in\_assign\_seq | 1/1 |
| task\_history\_no | Task\_history | t\_h\_no\_in\_th\_seq | 1/1 |
| client\_no | Client | c\_no\_in\_c\_seq | 1/1 |
| invoice\_no | Invoice | i\_no\_in\_i\_seq | 1/1 |
| expense\_id | Expense | exp\_id\_in\_exp\_seq | 1/1 |
| session\_no | Sessions | sess\_no\_in\_sess\_seq | 1/1 |
|  |  | ch\_no\_in\_ch\_seq | 1/1 |

Stored Programme Units

*This section should describe any stored functions procedures or triggers that you believe would be helpful. You should also give the reasons. For example*

|  |  |  |
| --- | --- | --- |
| Function / Procedure/ Trigger | Name / description | Rationale |
| Function | get\_estimate\_hours(task\_id) | Simplifies process of identifying the estimated\_hours applicable at the date of an estimate. Useful when re-calculating estimates or calculating capped price invoices. |
| Trigger | update\_estimated\_hours  (pre-update on task estimated\_hours) | Ensures that all changes to est\_hours in task are automatically recorded in order to avoid errors in recalculating estimates or calculating capped price invoices |

***Note: it is not anticipated that you will be able to identify many of these at this stage, in reality since we have not yet covered procedures and triggers you will normally be considering only any functions you think would be useful.***

Other Physical Design Issues

*If you would also like to include a section on useful* ***views or any other physical design issues****, please feel free to do so but this is not explicitly required.*

ASSUMPTIONS MADE

***Please state all assumptions made.*** *Whilst the information you have been given in the case study document is limited so that you may have to make some assumptions, you should check that none of your assumptions contradicts anything that is in the case study document.*

# Appendices

## Integrity

Concurrency

1)

A transaction comprises a unit of work performed within a [database management system](http://en.wikipedia.org/wiki/Database_management_system) (or similar system) against a database, and treated in a coherent and reliable way independent of other transactions.

Transactions in a database have two main purposes:

1. To provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure, when execution stops (completely or partially) and many operations upon a database remain uncompleted, with unclear status.
2. To provide isolation between programs accessing a database concurrently. If this isolation is not provided, the program's outcome are possibly erroneous.

A database transaction, by definition, must be atomic, consistent, isolated and durable, also known as ACID.

A transaction have an all-or-nothing proposition, which means that each work-unit performed must either complete in its entirety or have no effect at all. This means that if a transaction only reaches half completion and the system fails, the transaction is cancelled to prevent incorrect data being produced and saved. The system must also isolate each transaction from other transactions and results must conform to existing constraints in the database.

Transactions are important as they maintain the integrity of the data. A single transaction consists of one or more independent units of work, each reading and/or writing information to a database or other data store. When this occurs it is important to ensure that all such processing leaves the database in a consistent state.

Another property of transactions, isolation, is very important too. The isolation property defines how/when the changes made by one person become visible to other concurrent operations. An example would be someone depositing £100 into their bank account and also withdrawing £90. The system needs to complete two transactions, the depositing of £100 and only when that has finished the withdrawal of £90. This is to stop incorrect data being saved.

2)

A consistent state is used to describe the data’s validity in your system, and consistency as a property in databases is very important. It ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including but not limited to constraints, cascades, triggers, and any combination thereof. Every transaction must leave the database in a consistent (correct) state. It must transform a database from one consistent state to another consistent state. Thus since a database can be normally changed only by transactions, all the database's states are consistent. An aborted transaction does not change the database state it has started from, as if it never existed.

3)

When a database commits, it is making changes permanent. It is commonly used at the end of a transaction. The commit statement makes all changes visible to other users of the database.

Rollback is the process of undoing a changed to a database. This is useful if incorrect information has been entered into the database and you need to remove it.

A savepoint is a way of implementing a subtransaction (also known as nested transactions) within a RDMS by indicating a point within a transaction that can be “rolled back to” without affecting any work done in the transaction before the savepoint was created. More than one savepoint can exist within a single transaction. Savepoints are useful for implementing complex error recovery in database application – if an error occurs in the midst of a multiple-statement transaction, the application may be able to recover from the error without needing to abort the entire transaction.

So together commit, rollback and savepoints ensure that data integrity is enforced throughout the database system. If an incorrect transaction has commit, it can easily be rolled back to restore the database to its previous state. And if a complex transaction is taking place, savepoints can be used at key points of the transaction so that if an error occurs such as power failure, the whole transaction will not need to be performed again and instead will resume from the last savepoint.

4)

Optimistic concurrency control assumes that multiple transactions can frequently complete without interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.

Pessimistic concurrency control assumes that two or more users will want to update the same record at the same time, and then prevents that possibility by locking the record, no matter how unlikely conflicts actually are. The locks are placed as soon as any piece of the row is accessed, making it impossible for two or more users to update the row at the same time. Depending on the lock mode (shared, exclusive or update), other users might be able to read the date even though a lock has been placed.

As for when each should be used, if you are dealing with a database that is constantly being edited the Pessimistic concurrency control may be better suited, if conflicts will not be an issue then Optimistic concurrency would be better suited as it is faster and simpler to work with.

5)

If someone is reading from a database at the same time as someone else is writing to it, it is possible that the reader will see a half-written or inconsistent piece of data. There are different control methods to get around this issue, and I will explain two, locking and multiversions.

Locking is a method that forces the readers to wait until the writers are finished updating a row for example. This method is very slow and prevents users from accessing part of the database if someone else is already accessing it. This is the pessimistic control approach. The primary advantage is that no other user can get a lock on the record for updating, effectively informing any requesting user that they cannot update the record because it is in use. Since web applications can have hundreds or thousands of simultaneous users, a persistent connection to the database cannot be maintained without having tremendous resources on the database server. There are several drawbacks to locking. If the user requests a record, and the lock is placed on it, and the user then leaves the computer for any period of time, the record will remain locked and deny access to anyone else so long as the current user has a lock on it. In order to maintain record locks, a persistent connection to the database server is required.

Multiversion concurrency control is a method commonly used by database management systems to provide concurrent access to the database. Each user connected to the database sees a snapshot of the database at a particular instant in time. Any changes made by a writer will not be seen by other users of the database until the changes have been completed. When a MVCC database needs to update an item of data, it will not overwrite the old data with new data, but instead mark the old data as obsolete and add the newer version elsewhere. Thus there are multiple versions store, but only one is the latest. This allows readers to access data that was there when they began reading, even if it was modified or deleted part way through by someone else. It allows the database to avoid the overhead of filling in holes in memory or disk structures but requires the system to periodically sweep through and delete the old, obsolete data objects. MVCC provides point in time consistent views. Read transactions under MVCC typically use a timestamp or transaction ID to determine what state of the database to read, and read those versions of the data. This avoids managing locks for read transactions because writes can be isolated by virtue of the old versions being maintained, rather than through a process of locks or mutexes. Writes affect a future version but at the transaction ID that the read is working at, everything is guaranteed to be consistent because the writes are occurring at a later transaction ID.

<http://en.wikipedia.org/wiki/Multiversion_concurrency_control>

http://en.wikipedia.org/wiki/Optimistic\_concurrency\_control

<http://en.wikipedia.org/wiki/Concurrency_control>

http://www.cslab.uky.edu/apps/odocs/osji/apiug/6n\_mvcc.htm

https://devcenter.heroku.com/articles/postgresql-concurrency

Figure 1 – David’s work on concurrency.

Questions.

6. The term ‘level of isolation’ means the degree to which one transaction is isolated from resource or data modifications made by other transactions. It is also described in terms of which concurrency side-effects, such as dirty reads or phantom reads, are allowed.

7.. These categories of phenomena mean that isolation levels are defined in terms of three phenomena that must be prevented between concurrently executing transactions.

The three preventable phenomena are:

Dirty reads: A transaction reads data which has written by another transaction that has not been committed yet.

Nonrepeatable reads: A transaction rereads data it previously read and finds that another committed transaction has modified or deleted the data.

Phantom reads (or phantoms): A transaction re-runs a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

8. Oracle offers the read committed and serializable isolation levels, as well as a read-only mode that is not part of SQL92. Read committed is the default.

9. Statement-level read consistency guarantees that all the data returned by a single query comes from a single point in time and the time that the query began. So a query never sees dirty data or any of the changes made by transactions that commit during query execution. As query execution proceeds, only data committed before the query began is visible to the query. The query does not see changes committed after statement execution begins.

Transaction-level read consistency is when a transaction runs in serializable mode, all data accesses reflect the state of the database as of the time the transaction began. This means that the data seen by all queries within the same transaction is consistent with respect to a single point in time, except that queries made by a serializable transaction do see changes made by the transaction itself. Transaction-level read consistency produces repeatable reads and does not expose a query to phantoms.

10. By having READ COMMITTED and SERIALIZABLE Isolation because both levels provide the contention-reducing benefits of Oracle's "read consistency" multi-version concurrency control model and exclusive row-level locking implementation, and are designed for real-world application deployment.

Figure 2 – Tinashes work on Concurrency

-- File name ProSOftCreateExtended

-- Purpose To populate the extended ProSoft Project Management System (PMS) database

-- with test data (includes tables client, invoice, expense, sessions and client\_history

-- Language SQL (Oracle)

-- Orig Author Sheila Baron

-- Date October 2009

-- Changes Sept 2013: addition to TO\_DATE wrapper around all date values to

-- accomodate different default date format in Apex (MM-DD-YY, eg 09-30-13)

-- Further revisions add test data for four new tables

-- Version 3.0

-- N.B. need to run sequences script before running this file

-- first remove any records already in the tables

DELETE FROM sessions;

DELETE FROM expense;

DELETE FROM client\_history;

DELETE FROM task\_history;

DELETE FROM assignment;

DELETE FROM task;

DELETE FROM stage;

DELETE FROM project;

DELETE FROM employee;

DELETE FROM task\_type;

DELETE FROM charge\_grade;

DELETE FROM client;

DELETE FROM invoice;

-- data represents: client\_no, company\_name, address\_1, address\_2, town, county, postcode, contact\_name, tel\_no, fax\_no, email

INSERT INTO client VALUES (c\_no\_in\_c\_seq.NEXTVAL, 'Barco Ltd', 'Venture House', 'Downshire Way', 'Bracknell', 'Berkshire', 'RG12 1WA', 'Barry Smith', 08082348122, 01619998888, 'Service.HR@barco.com');

INSERT INTO client VALUES (c\_no\_in\_c\_seq.NEXTVAL, 'MySports', '23 Cattistock Road', 'Mottingham', 'London', 'London', 'SE9 4AW', 'Sarah Wilson', 07737984496, 01619638754, 'info@my-sports.co.uk');

-- data represents: invoice\_no, invoice\_date, due\_date, paid\_date

INSERT INTO invoice VALUES (i\_no\_in\_i\_seq.NEXTVAL, TO\_DATE('10-may-09','DD-MON-YY'), TO\_DATE('17-may-09','DD-MON-YY'), TO\_DATE('16-may-09','DD-MON-YY'));

INSERT INTO invoice VALUES (i\_no\_in\_i\_seq.NEXTVAL, TO\_DATE('18-may-09','DD-MON-YY'), TO\_DATE('23-may-09','DD-MON-YY'), TO\_DATE('22-may-09','DD-MON-YY'));

INSERT INTO invoice VALUES (i\_no\_in\_i\_seq.NEXTVAL, TO\_DATE('24-may-09','DD-MON-YY'), TO\_DATE('29-may-09','DD-MON-YY'), TO\_DATE('28-may-09','DD-MON-YY'));

-- data represents grade\_no, description, rate\_per\_hour

INSERT INTO charge\_grade VALUES (1,'Director',50.0);

INSERT INTO charge\_grade VALUES (2,'Senior Engineer',40.0);

INSERT INTO charge\_grade VALUES (3,'Engineer',20.0);

-- data represents: employee\_no, firstname, lastname, grade\_no

INSERT INTO employee VALUES (1,'John','Smith',3);

INSERT INTO employee VALUES (2,'Peter','White',2);

-- data represents: task\_type\_no, description, grade\_no

INSERT INTO task\_type VALUES (10,'Discuss proposal',1);

INSERT INTO task\_type VALUES (20,'Analyse overall',2);

INSERT INTO task\_type VALUES (31,'Design in detail',3);

INSERT INTO task\_type VALUES (32,'Produce documentation',3);

INSERT INTO task\_type VALUES (40,'Software Development',2);

INSERT INTO task\_type VALUES (53,'Install system',2);

INSERT INTO task\_type VALUES (54,'Test system',3);

INSERT INTO task\_type VALUES (99,'Other',NULL);

-- data represents: project\_no, project\_code, description, client\_no, mgr\_no

INSERT INTO project VALUES (1,'MISTEST', 'HR System Development', 1,1);

INSERT INTO project VALUES (2,'ECOM\_MS', 'E-commerce System', 2,1);

-- data represents: stage\_id, project\_no, stage\_no, description,

-- start\_date, planned\_duration, fees\_basis, estimate\_date

INSERT INTO stage VALUES (1, 1, 1,'Development of system', TO\_DATE('08-MAY-09','DD-MON-YY'), 40, 'VS', TO\_DATE('17-JAN-09','DD-MON-YY'));

INSERT INTO stage VALUES (2, 1, 2,'Testing of system', TO\_DATE('15-JUL-09','DD-MON-YY'), 40, 'VS', TO\_DATE('17-JAN-09','DD-MON-YY'));

INSERT INTO stage VALUES (3, 2, 1,'Rquirements Analysis', TO\_DATE('10-JAN-10','DD-MON-YY'), 40, 'VS', TO\_DATE('17-JUL-10','DD-MON-YY'));

-- data represents: task\_id, task\_no, description, estimated\_hours,

-- planned\_start\_date, max\_duration, stage\_id, task\_type\_no

INSERT INTO task VALUES (1,1, 'Discuss proposal with HCC',8.0, TO\_DATE('08-may-09','DD-MON-YY'),2, 10,1);

INSERT INTO task VALUES (2,2, 'Analyse system', 80.0, TO\_DATE('12-may-09','DD-MON-YY'),3, 20,1);

INSERT INTO task VALUES (3,3, 'Design database', 5.0, TO\_DATE('23-may-09','DD-MON-YY'),1, 31,1);

INSERT INTO task VALUES (4,4, 'Produce draft Req.Spec.', 20.0, TO\_DATE('24-may-09','DD-MON-YY'),2, 32,1);

INSERT INTO task VALUES (5,5, 'Build system', 600.0, TO\_DATE('04-jun-09','DD-MON-YY'),12,40,1);

INSERT INTO task VALUES (6,6, 'Test system modules', 100.0, TO\_DATE('15-jun-09','DD-MON-YY'),12,54,1);

INSERT INTO task VALUES (7,1, 'Integrate with existing systems',100.0,TO\_DATE('01-Oct-09','DD-MON-YY'),15,99,2);

INSERT INTO task VALUES (8,2, 'Integration testing', 45.0, TO\_DATE('29-Oct-09','DD-MON-YY'), 3,54,2);

-- data represents: employee\_no, task\_id, estimated\_hours, completed date

INSERT INTO assignment VALUES (1,1,1, 8, TO\_DATE('12-May-09','DD-MON-YY'));

INSERT INTO assignment VALUES (2,1,2, 50, TO\_DATE('22-may-09','DD-MON-YY'));

INSERT INTO assignment VALUES (3,2,2, 30, TO\_DATE('23-May-09','DD-MON-YY'));

INSERT INTO assignment VALUES (4,1,3, 5, TO\_DATE('24-May-09','DD-MON-YY'));

INSERT INTO assignment VALUES (5,1,4, 20, TO\_DATE('04-Jun-09','DD-MON-YY'));

INSERT INTO assignment VALUES (6,1,5, 400,TO\_DATE('27-aug-09','DD-MON-YY'));

INSERT INTO assignment VALUES (7,2,5, 200,TO\_DATE('15-aug-09','DD-MON-YY'));

INSERT INTO assignment VALUES (8,2,6, 100,TO\_DATE('21-sep-09','DD-MON-YY'));

INSERT INTO assignment VALUES (9,1,7, 100, NULL);

INSERT INTO assignment VALUES (10,2,8, 45, NULL);

-- data represents: task\_history\_id, previous\_est\_hours, change\_date,

-- reason\_for\_change, change\_userid, task\_id

INSERT INTO task\_history VALUES (1, 10.0, TO\_DATE('28-may-09','DD-MON-YY'), 'original estimate unrealisable', 'BARONS',4);

INSERT INTO task\_history VALUES (2, 500.0, TO\_DATE('10-jul-09','DD-MON-YY'), 'original estimate unrealisable', 'BARONS',5);

INSERT INTO task\_history VALUES (3, 650.0, TO\_DATE('15-aug-09','DD-MON-YY'), 'have been able to trim a little time','BARONS',5);

INSERT INTO task\_history VALUES (4, 250.0, TO\_DATE('08-aug-09','DD-MON-YY'), 'test plans extended','BARONS', 6);

-- data represents: expense\_id, description, quantity, unit\_cost, chargeable, project\_no, invoice\_no

INSERT INTO expense VALUES (exp\_id\_in\_exp\_seq.NEXTVAL, 'Folder', 3, 5, 'Y', 1, null);

INSERT INTO expense VALUES (exp\_id\_in\_exp\_seq.NEXTVAL, 'Tool', 1, 100, 'Y', 2, null);

INSERT INTO expense VALUES (exp\_id\_in\_exp\_seq.NEXTVAL, 'Disks', 5, 2, 'N', 1, null);

-- data represents: session\_no, session\_date, actual\_hours, chargeable, assignment\_no, invoice\_no

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('11-may-09','DD-MON-YY'), 8, 'Y', 1, 1);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('16-may-09','DD-MON-YY'), 2, 'Y', 2, 1);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('22-may-09','DD-MON-YY'), 1, 'N', 2, 2);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('22-may-09','DD-MON-YY'), 8, 'Y', 2, 2);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('09-may-09','DD-MON-YY'), 4, 'Y', 3, 1);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('09-may-09','DD-MON-YY'), 2, 'Y', 3, 1);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('12-may-09','DD-MON-YY'), 4, 'Y', 4, 1);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('13-may-09','DD-MON-YY'), 3, 'Y', 4, 1);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('16-may-09','DD-MON-YY'), 1, 'Y', 4, 1);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('20-may-09','DD-MON-YY'), 10, 'Y', 4, 2);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('21-may-09','DD-MON-YY'), 40, 'Y', 5, 2);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('21-may-09','DD-MON-YY'), 6, 'N', 5, 2);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('28-may-09','DD-MON-YY'), 30, 'Y', 5, 3);

INSERT INTO sessions VALUES (sess\_no\_in\_sess\_seq.NEXTVAL, TO\_DATE('28-may-09','DD-MON-YY'), 40, 'Y', 5, 3);

-- save the data

commit;

-- check consistency of dates and values

select p.project\_no, s.stage\_no, t.task\_no, t.planned\_start\_date,

t.max\_duration, t.estimated\_hours, e.lastname, a.estimated\_hours, a.completed\_date

FROM project p, stage s, task t, assignment a, employee e

WHERE p.project\_no = s.project\_no

and s.stage\_id = t.stage\_id

and t.task\_id = a.task\_id

and e.employee\_no = a.employee\_no

order by p.project\_no, s.stage\_no, t.task\_no;

-- finally view data set generated

select \* from charge\_grade;

select \* from task\_type;

select \* from employee;

select \* from project;

select \* from stage;

select \* from task;

select \* from assignment;

select \* from task\_history;

Figure 3 – Mark and Kudzai’ work on PopulateProSoftExtended

-- program header needed

-- updated to include client\_history sequence + drop sequence statements - Maziva 31-3-14

DROP SEQUENCE g\_no\_in\_cg\_seq;

DROP SEQUENCE tt\_no\_in\_tt\_seq;

DROP SEQUENCE emp\_no\_in\_emp\_seq;

DROP SEQUENCE p\_no\_in\_p\_seq;

DROP SEQUENCE p\_c\_in\_p\_seq;

DROP SEQUENCE s\_id\_in\_s\_seq;

DROP SEQUENCE s\_no\_in\_s\_seq;

DROP SEQUENCE t\_id\_in\_task\_seq;

DROP SEQUENCE t\_no\_in\_t\_seq;

DROP SEQUENCE a\_no\_in\_assign\_seq;

DROP SEQUENCE t\_h\_no\_in\_th\_seq;

DROP SEQUENCE c\_no\_in\_c\_seq;

DROP SEQUENCE i\_no\_in\_i\_seq;

DROP SEQUENCE exp\_id\_in\_exp\_seq;

DROP SEQUENCE sess\_no\_in\_sess\_seq;

DROP SEQUENCE ch\_no\_in\_ch\_seq;

CREATE SEQUENCE g\_no\_in\_cg\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE tt\_no\_in\_tt\_seq

START WITH 10 INCREMENT by 10;

CREATE SEQUENCE emp\_no\_in\_emp\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE p\_no\_in\_p\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE p\_c\_in\_p\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE s\_id\_in\_s\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE s\_no\_in\_s\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE t\_id\_in\_task\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE t\_no\_in\_t\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE a\_no\_in\_assign\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE t\_h\_no\_in\_th\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE c\_no\_in\_c\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE i\_no\_in\_i\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE exp\_id\_in\_exp\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE sess\_no\_in\_sess\_seq

START WITH 1 INCREMENT by 1;

CREATE SEQUENCE ch\_no\_in\_ch\_seq

START WITH 1 INCREMENT by 1;

Figure 4 – Mark and Kudzai’ work on ProSoftSequences

-- program header needed

-- date of last revision 31.3.14

-- version 4.0

CREATE OR REPLACE FUNCTION isValidPostCode

(postcode VARCHAR2)

RETURN NUMBER

IS

BEGIN

IF (postcode IS NOT NULL

AND (Regexp\_like(postcode,'[ABCDEFGHIJKLMNOPRSTUWYZ]{1}[0-9]{1} [0-9]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}','i')

OR Regexp\_like(postcode,'[ABCDEFGHIJKLMNOPRSTUWYZ]{1}[0-9]{1}[0-9]{1} [0-9]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}','i')

OR Regexp\_like(postcode,'[ABCDEFGHIJKLMNOPRSTUWYZ]{1}[ABCDEFGHKLMNOPQRSTUVWXY]{1}[0-9]{1} [0-9]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}','i')

OR Regexp\_like(postcode,'[ABCDEFGHIJKLMNOPRSTUWYZ]{1}[ABCDEFGHKLMNOPQRSTUVWXY]{1}[0-9]{1}[0-9]{1} [0-9]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}','i')

OR Regexp\_like(postcode,'[ABCDEFGHIJKLMNOPRSTUWYZ]{1}[0-9]{1}[ABCDEFGHJKSTUW]{1} [0-9]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}','i')

OR Regexp\_like(postcode,'[ABCDEFGHIJKLMNOPRSTUWYZ]{1}[ABCDEFGHKLMNOPQRSTUVWXY]{1}[0-9]{1}[ABEHMNPRVWXY]{1} [0-9]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}[ABDEFGHJLNPQRSTUWXYZ]{1}','i'))) THEN

RETURN 1;

ELSE

RETURN 0;

END IF;

END isValidPostCode;

/

CREATE OR REPLACE FUNCTION isValidEmail

(email VARCHAR2)

RETURN NUMBER

IS

BEGIN

IF (email IS NOT NULL

AND (REGEXP\_LIKE(email, '[a-zA-Z0-9.\_%-]+@[a-zA-Z0-9.\_%-]+\.[a-zA-Z]{2,4}')

)) THEN

RETURN 1;

ELSE

RETURN 0;

END IF;

END isValidEmail;

/

-- First drop any existing tables

DROP TABLE client\_history;

DROP TABLE sessions;

DROP TABLE expense;

DROP TABLE task\_history;

DROP TABLE assignment;

DROP TABLE task;

DROP TABLE stage;

DROP TABLE project;

DROP TABLE employee;

DROP TABLE task\_type;

DROP TABLE charge\_grade;

DROP TABLE client;

DROP TABLE invoice;

-- Then create the current version

CREATE TABLE client

(

client\_no NUMBER(4),

company\_name VARCHAR2(50),

address\_1 VARCHAR2(50),

address\_2 VARCHAR2(50),

town VARCHAR2(50),

county VARCHAR2(50),

postcode VARCHAR2(8),

contact\_name VARCHAR2(50),

tel\_no NUMBER(11),

fax\_no NUMBER(11),

email VARCHAR2(50),

CONSTRAINT client\_pk PRIMARY KEY (client\_no),

CONSTRAINT address1\_nn CHECK (address\_1 IS NOT NULL),

CONSTRAINT town\_nn CHECK (town IS NOT NULL),

CONSTRAINT contact\_name\_nn CHECK (contact\_name IS NOT NULL),

CONSTRAINT tel\_no\_nn CHECK (tel\_no IS NOT NULL),

CONSTRAINT email\_uk UNIQUE(email),

--CONSTRAINT postcode\_enn CHECK(isValidPostCode(postcode)>0),

CONSTRAINT email\_f\_check CHECK (REGEXP\_LIKE(email, '[a-zA-Z0-9.\_%-]+@[a-zA-Z0-9.\_%-]+\.[a-zA-Z]{2,4}'))

);

-- cannot call user defined functions in a check constraint which the postcode\_enn did - see code above

-- as did the email\_f\_check

CREATE TABLE client\_history

(

client\_hist\_no NUMBER (4),

client\_no NUMBER(4),

company\_name VARCHAR2(50),

address\_1 VARCHAR2(50),

address\_2 VARCHAR2(50),

town VARCHAR2(50),

county VARCHAR2(50),

postcode VARCHAR2(8),

contact\_name VARCHAR2(50),

tel\_no NUMBER(11),

fax\_no NUMBER(11),

email VARCHAR2(50),

userid VARCHAR2(8),

change\_date date,

change\_reason varchar2(200),

CONSTRAINT client\_hist\_pk PRIMARY KEY (client\_hist\_no),

CONSTRAINT client\_hist\_client\_FK FOREIGN KEY (client\_no) REFERENCES client(client\_no)

);

CREATE TABLE invoice

(

invoice\_no NUMBER(2),

invoice\_date DATE DEFAULT sysdate,

due\_date DATE,

paid\_date DATE,

CONSTRAINT invoice\_pk PRIMARY KEY (invoice\_no),

CONSTRAINT invoice\_date\_nn CHECK (invoice\_date IS NOT NULL),

CONSTRAINT due\_date\_nn CHECK (due\_date IS NOT NULL),

CONSTRAINT invoice\_valid\_nnc CHECK (due\_date>invoice\_date),

CONSTRAINT invoice\_payment\_nnc CHECK (paid\_date>invoice\_date)

);

CREATE TABLE charge\_grade

(

grade\_no NUMBER(2),

description VARCHAR2(50),

hrly\_charge\_rate NUMBER(5,2),

CONSTRAINT charge\_grade\_pk PRIMARY KEY(grade\_no),

CONSTRAINT hrly\_charge\_rate\_nn CHECK (hrly\_charge\_rate>0)

);

CREATE TABLE task\_type

(

task\_type\_no NUMBER(2),

description VARCHAR2(50),

grade\_no NUMBER(2),

CONSTRAINT task\_type\_pk PRIMARY KEY (task\_type\_no),

CONSTRAINT task\_type\_charge\_grade FOREIGN KEY (grade\_no)

REFERENCES charge\_grade (grade\_no)

);

CREATE TABLE employee

(

employee\_no NUMBER(2),

firstname VARCHAR2(50),

lastname VARCHAR2(50),

grade\_no NUMBER(2),

CONSTRAINT employee\_pk PRIMARY KEY (employee\_no),

CONSTRAINT firstname\_nn CHECK (firstname IS NOT NULL),

CONSTRAINT lastname\_nn CHECK (lastname IS NOT NULL),

CONSTRAINT employee\_charge\_grade\_fk FOREIGN KEY (grade\_no)

REFERENCES charge\_grade (grade\_no)

);

CREATE TABLE project

(

project\_no NUMBER(2),

project\_code VARCHAR2(8),

description VARCHAR2(50),

client\_no NUMBER(2),

mgr\_no NUMBER(2),

CONSTRAINT project\_pk PRIMARY KEY (project\_no),

CONSTRAINT project\_code\_nn CHECK (project\_code IS NOT NULL),

CONSTRAINT project\_employee\_fk FOREIGN KEY (mgr\_no)

REFERENCES employee(employee\_no),

CONSTRAINT project\_client\_fk FOREIGN KEY (client\_no)

REFERENCES client (client\_no)

);

-- CREATE UNIQUE INDEX project\_code\_index

-- ON project(project\_no);

--/

CREATE TABLE expense

(

expense\_id NUMBER(2),

description VARCHAR2(50),

quantity NUMBER(4),

unit\_cost NUMBER(6,2),

chargeable VARCHAR2(1),

project\_no NUMBER(2),

invoice\_no NUMBER(2),

CONSTRAINT expense\_pk PRIMARY KEY (expense\_id),

CONSTRAINT expense\_chargeable\_nn CHECK (chargeable in ('Y','N')),

CONSTRAINT expense\_project\_fk FOREIGN KEY (project\_no)

REFERENCES project(project\_no),

CONSTRAINT expense\_invoice\_fk FOREIGN KEY (invoice\_no)

REFERENCES invoice (invoice\_no)

);

-- planned duration is expressed as a number of weeks

-- fees-basis can take only the following values VS, VN, CS, CN, FQ

CREATE TABLE stage

(

stage\_id NUMBER(2),

project\_no NUMBER(2),

stage\_no NUMBER(2),

description VARCHAR2(50),

start\_date DATE,

planned\_duration NUMBER(2),

fees\_basis VARCHAR2(2),

estimate\_date DATE,

CONSTRAINT stage\_pk PRIMARY KEY (stage\_id),

CONSTRAINT stage\_project\_fk FOREIGN KEY (project\_no)

REFERENCES project (project\_no),

CONSTRAINT p\_no\_in\_stage\_nn CHECK (project\_no IS NOT NULL),

CONSTRAINT p\_fees\_nn CHECK (fees\_basis IS NOT NULL),

CONSTRAINT fees\_basis\_nn CHECK (fees\_basis in ('VS','VN','CS','CN','FQ'))

);

CREATE TABLE task

(

task\_id NUMBER(2),

task\_no NUMBER(2),

description VARCHAR2(50),

estimated\_hours NUMBER(6,1) ,

planned\_start\_date DATE,

max\_duration NUMBER(2),

task\_type\_no NUMBER(2),

stage\_id NUMBER(2),

CONSTRAINT task\_pk PRIMARY KEY (task\_id),

CONSTRAINT task\_stage\_fk FOREIGN KEY (stage\_id)

REFERENCES stage(stage\_id),

CONSTRAINT task\_tt\_fk FOREIGN KEY (task\_type\_no)

REFERENCES task\_type (task\_type\_no),

CONSTRAINT s\_id\_in\_task\_nn CHECK(stage\_id IS NOT NULL)

);

CREATE TABLE assignment

(

assignment\_no NUMBER(2),

employee\_no NUMBER(2),

task\_id NUMBER(2),

estimated\_hours NUMBER(6,1),

completed\_date DATE,

CONSTRAINT assignment\_pk PRIMARY KEY (assignment\_no),

CONSTRAINT assignment\_employee\_fk FOREIGN KEY (employee\_no)

REFERENCES employee(employee\_no),

CONSTRAINT assignment\_task\_fk FOREIGN KEY (task\_id)

REFERENCES task(task\_id)

);

CREATE TABLE task\_history

(

task\_history\_id NUMBER(2),

prev\_est\_hrs NUMBER(6,1),

change\_date DATE DEFAULT sysdate,

reason\_for\_change LONG,

change\_userid VARCHAR2(50),

task\_id NUMBER(2),

CONSTRAINT task\_history\_PK PRIMARY KEY (task\_history\_id),

CONSTRAINT prev\_est\_nn CHECK(prev\_est\_hrs IS NOT NULL),

CONSTRAINT task\_history\_task\_fk FOREIGN KEY (task\_id)

REFERENCES task(task\_id)

);

CREATE TABLE sessions

(

session\_no NUMBER(2),

session\_date DATE,

actual\_hrs NUMBER(6,1),

chargeable VARCHAR2(1),

assignment\_no NUMBER(2),

invoice\_no NUMBER(2),

CONSTRAINT session\_pk PRIMARY KEY (session\_no),

CONSTRAINT actuak\_hrs\_nn CHECK(actual\_hrs IS NOT NULL),

CONSTRAINT s\_chargable\_nn CHECK(chargeable IS NOT NULL),

CONSTRAINT s\_chargeable\_nn CHECK (chargeable in ('Y','N')),

CONSTRAINT s\_assignment\_fk FOREIGN KEY (assignment\_no)

REFERENCES assignment(assignment\_no),

CONSTRAINT s\_invoice\_fk FOREIGN KEY (invoice\_no)

REFERENCES invoice (invoice\_no)

);

Figure 5 – Mark and Kudzai’ work on new tested precreate tables

## Security

Access Control Task

Use this form to specify access control requirements for all roles in the ProSoft Project Management System

You will need to think through what views are appropriate before you define the table privileges required. This will involve performing a detailed analysis of the ProSoft case study.

Use the following codes to indicate privileges required:

S = SELECT,

I = INSERT,

U = UPDATE,

D = DELETE,

E = EXECUTE (applies to procedures and functions only)

Please add as many rows as you need to the various sections of this table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Database Object** | **Project Manager Role** | **Engineer Role** | **Sales**  **Role** | **Personnel Role** | **Accounts Role** |  |
| **TABLES** | | | | | | |
| Charge\_Grade | S |  | S | SIUD | SIUD |  |
| Task-Type | SIUD |  |  | S | SIUD |  |
| Client | SIUD |  | SIUD | SU | S |  |
| Project | SIUD |  | S | S | S |  |
| Stage | SIUD |  | S | S | SU |  |
| Task | SIUD |  | S | S | S |  |
| Task History | S | This table populated by a trigger apart from reason\_for\_change which is updated by a procedure | | | |  |
| Employee | S |  |  | SIUD |  |  |
| Assignment | SIUD |  |  |  |  |  |
| Sessions\* | SIUD |  | S |  |  |  |
| Expense\* | SIUD |  |  |  |  |  |
| Invoice | S |  | S |  | S |  |
|  |  |  |  |  |  |  |
| SEQUENCES | | | | | | |
| Charge\_Grade |  |  | S |  | S |  |
| Task-Type | S |  |  |  | S |  |
| Client | S |  | S |  |  |  |
| Project | S |  | S |  |  |  |
| Stage | S |  | S |  |  |  |
| Task | S |  | S |  |  |  |
| Task History | This sequences used by a trigger | | | | |  |
| Employee |  |  |  |  |  |  |
| Assignment | S |  |  |  |  |  |
| Sessions | S | S | S |  |  |  |
| Expense | S |  | S |  |  |  |
| Invoice\_seq | This sequence used by procedure create\_invoice\_records | | | | |  |
|  |  |  |  |  |  |  |
| VIEWS (see also view definitions below) | | | | | | |
| Project\_descriptions | S | S |  | S |  |  |
| Own\_task\_assignments |  | S |  |  |  |  |
| Own\_sessions |  | SIU |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| FUNCTIONS | | | | | | |
| Estimate\_task\_cost | E |  | E |  | E |  |
| Estimate\_stage\_cost | E |  | E |  | E |  |
|  |  |  |  |  |  |  |
| PROCEDURES | | | | | | |
| Update\_estimate\_date | E |  | E |  | E |  |
| Create\_invoice\_records | E |  | E |  |  |  |
| Update\_th\_reason | E |  | E |  |  |  |
|  |  |  |  |  |  |  |
| VIEW DEFINITIONS: | | | | | | |
| Name: PROJECT\_DESCRIPTIONS  Contains: mgr.last\_name||’, ‘||first\_name manager, project\_no, project\_code, p\_description, stage\_no, stage\_descrition, task\_no , task\_description, task\_id (for joins) task\_start\_date, task\_max\_duration, task\_estimated\_hours. | | | | | | |
| Name: OWN\_TASK\_ASSIGNMENTS  Contains: All columns from assignment plus firstname, lastname from employee. Restricted to assignments relating to tasks where the employee has been assigned work. Ie the user can see his/her own assignments plus other assignments for the tasks they are working on. | | | | | | |
| Name: OWN\_SESSIONS  Contains: all columns from sessions table restricted to records relating to the current user. | | | | | | |
| Name: Own\_charge\_grade  Contains:grade\_no from EMPLOYEE | | | | | | |
|  | | | | | | |
| ASSUMPTIONS MADE:  Only project managers can create new estimates and invoices because they are the only ones who know when the project data is sufficiently complete / up-to-date to do this. Sales and Accounts can see details of all existing estimates and invocies.  Triggers will prevent updates and deletes to any session or expense record that has already been included on an invoice.  David:  I assume the engineer shouldn’t be allowed to view his/her pay grade & the other employees. I also assume the engineer doesn’t have projects he/she is linked to so views for projects and stages seem unnecessary .  Mark:  For project managers I assumed they’d have full control over most table except for anything linked to pay, task history and clients. All of which can be selected though to view.  For accounts I assumed that tables Client,Project,Stage,Task and Charge-grade needed to be selectable in order to create estimates. This also means the Stage table will need to be updated to record the date of the estimate. | | | | | | |

Figure 6 shows the user roles table completed by David, Mark and Tinashe

-- File name Create views script

-- Purpose To create the views specified in the initial analysis and access requirements of the ProSoft Project Management System (PMS) database

-- Language SQL (Oracle)

-- Orig Author Mark Tomlin

-- Date 12 May 2014

-- Version 1.0

CREATE OR REPLACE VIEW project\_descriptions AS

SELECT e.first\_name, e.surname, p.project\_no, p.project\_code, p.description, s.stage\_no, s.description, t.task\_no, t.description, t.planned\_start\_date, t.max\_duration, t.estimated\_hours

FROM employee e, project p, stage s, task t

WHERE e.employee\_no = p.mgr\_no

AND p.project\_no = s.project\_no

AND s.state\_id = t.stage\_id;

CREATE OR REPLACE VIEW own\_sessions AS

SELECT s.session\_no, s.session\_date, s.actual\_hours, s.chargeable, s.assignment\_no, s.invoice\_no

FROM employee e, assignment a, sessions s

WHERE e.employee\_no = a.employee\_no

AND a.assignment\_no = s.assignment\_no

AND e.user\_id = dbo.my\_userid\_function;

Figure 7 – Marks additional task showing his create views script

--trigger when client is changed

-- needs a program header

show errors;  
CREATE OR REPLACE TRIGGER audit\_client  
AFTER INSERT OR DELETE OR UPDATE ON client  
FOR EACH ROW  
  
BEGIN

INSERT INTO client\_history VALUES  
 (ch\_no\_in\_ch\_seq.NEXTVAL, :old.client\_no, :old.company\_name, :old.address\_1, :old.address\_2,:old.town,:old.county,:old.postcode, :old.contact\_name,:old.tel\_no,:old.fax\_no,:old.email, User, Sysdate, 'to be advised');

END;

Figure 8 shows Kudzai’ Triggers for task IV

## Performance

**PERFORMANCE ENHANCEMENT TOPIC – Writing good SQL Select statements**

The following document discusses ways in which the SQL Select statement can be improved upon. It is important to have efficient SQL statements, as it can optimise your system and prevent unnecessary workload for the database.

Using the “SELECT \*” statement can cause a few different problems. It can lead to the transfer of massive amounts of data that isn’t required and assuming the ProSoft database contains information on all of its employees and clients, this could slow down the system greatly if every statement used the SELECT ALL method (\*).

Another way to make the SELECT Statement more efficient is to only return data that will be used. This means not returning un-necessary data with a query. For example, using SELECT \* to return just the Employee number from EMPLOYEE, in the ProSoft case study, will also return the firstname, lastname and grade\_no. This is all irrelevant information if you only needed the Employee number. By selecting only the data you need, you can avoid unneeded overheads and time spent reading information by the system.

The use of indexing can improve a systems efficiency massively. Particularly large systems will benefit the most from indexing, as it provides a location to the data being requested, but can be processed far quicker than searching through the entire database.

Figure 9 shows the work contribution towards the report on Performance enhancement by David

**Most efficient table name sequence in a FROM clause**

When using a SELECT statement where the FROM clause specifies more than one table, the table containing the lowest number of rows should be placed last. This is because the Oracle parser will process the table names from right to left. When Oracle processes the state, the first table (last one listed) will merge/sort using an internal procedure with the second table. It has been proven that it is far more efficient to have the table with the lowest amount of rows sorted and merged first.

An example of this can be when selecting from the Prosoft Project and Stage tables.

Table Project has 2 rows and table Stage has 3 rows (different between numbers will only scale in future).

Therefore the most efficient way to write a SELECT statement from these tables is:

SELECT \* FROM Stage, Project;

If three tables are being joined you should the intersection table as the last specified table. The intersection table has the most tables dependent on it and is more efficient in this scenario.

**Combine simple unrelated SELECT statements together**

Rather than doing a list of simple select statements, it is far more efficient to combine them all into a single query. This can be done even when the original queries are unrelated and it will increase database performance.

Here is an example of list of SELECT queries:

SELECT grade\_no

FROM Employee

WHERE employee\_no = 1;

SELECT project\_code

FROM Project

WHERE project\_no = 1;

SELECT stage\_no

FROM Stage

WHERE stage\_id = 1;

These three statements can be combined together as shown below:

SELECT e.grade\_no, p.project\_code, s.stage\_no

FROM Employee e, Project p, Stage s, DUAL X

WHERE NVL('X', X.DUMMY) = NVL('X', e.ROWID (+))

AND NVL('X', X.DUMMY) = NVL('X', p.ROWID (+))

AND NVL('X', X.DUMMY) = NVL('X', s.ROWID (+))

AND e.employee\_no (+) = 1

AND p.project\_no (+) = 1

AND s.stage\_id (+) = 1

Figure 10 shows the work contribution to the Performance enhancement report by Mark

How and why the use of built in procedures and functions enhances database application performance

Stored procedures are only executed once and they stored in their executable form , so they are quick and efficient since they are already compiled so no need of recompilation .

Since they stored in the executable form , executable is automatically cached and shard among users , this lowers memory and invocation overhead there by enhancing performance .

Grouped sql statements with a stored procedure allows them executed with a single call , this minimises the use of slow networks ,reduces network traffic, and improves round trip response time .

Procedures can be moved from client to server ,where they will execute faster . This enhances a database application by executing application logic within the server.

Appendix

Oracle. (1999). Advantages of Stored Procedures. Available: http://docs.oracle.com/cd/F49540\_01/DOC/java.815/a64686/01\_intr3.htm. Last accessed 14 April 2014.

Figure 11 shows Kudzai’ additional work for task V.

**Position of Joins in the WHERE Clause**

The conditions of WHERE clause should come after the table joins are written. And the conditions which filter out the maximum records should be placed at the end after the joins as the parsing is done from BOTTOM to TOP.

For example

**Least Efficient:**

SELECT . . . . . . . . . . . .

FROM EMP E

WHERE SAL > 100,000

AND JOB = ‘MANAGER’

AND 15 < (SELECT COUNT (\*)

FROM EMP

WHERE MGR = E.EMPNO);

**Most Efficient:**

SELECT . . . .

FROM EMP E

WHERE 15 < (SELECT COUNT (\*)

FROM EMP

WHERE MGR = E.EMPNO)

AND SAL > 100,000

AND JOB = ‘MANAGER’;

**Use DECODE to Reduce Processing**

The DECODE statement provides a way to avoid having to scan the same rows repetitively, or to join the same table repetitively.

For example:

SELECT COUNT (\*), SUM (SAL)

FROM EMP

WHERE DEPT\_NO = 0020

AND ENAME LIKE ‘SMITH%’;

SELECT COUNT (\*), SUM (SAL)

FROM EMP

WHERE DEPT\_NO = 0030

AND ENAME LIKE ‘SMITH%’;

The same result can be achieved much more efficiently using DECODE:

SELECT COUNT (DECODE (DEPT\_NO, 0020, ‘X’, NULL)) D0020\_COUNT,

COUNT (DECODE (DEPT\_NO, 0030, ‘X’, NULL)) D0030\_COUNT,

SUM (DECODE (DEPT\_NO, 0020, SAL, NULL)) D0020\_SAL,

SUM (DECODE (DEPT\_NO, 0030, SAL, NULL)) D0030\_SAL

FROM EMP

WHERE ENAME LIKE ‘SMITH%’;

Similarly, DECODE can be used in GROUP BY or ORDER BY clause effectively.

**Use WHERE in Place of HAVING**

Avoid including a HAVING clause in SELECT statements. The HAVING clause filters selected rows only after all rows have been fetched. This could include sorting, summing, etc. Restricting rows via the WHERE clause, rather than the HAVING clause, helps reduce these overheads.

For example

**Least Efficient**:

SELECT REGION, AVG (LOC\_SIZE)

FROM LOCATION

GROUP BY REGION

HAVING REGION! = ‘DELHI’

AND REGION! = ‘NOIDA’

**Most Efficient:**

SELECT REGION, AVG (LOC\_SIZE)

FROM LOCATION

GROUP BY REGION

WHERE REGION! = ‘DELHI’

AND REGION! = ‘NOIDA’

Figure 12 shows the work contribution to the Performance enhancement report by Tinashe