Uni DB1 Syntax Details

Syntax details for the DB1 (databases) course at HdM Stuttgart

Felicitas Pojtinger

2022-02-01

“so basically i am monkey” - monke, monkeeee

Acknowledgements

Acknowledgements

Most of the following is based on the Oracle Tutorial.

Reset Everything

Reset Everything

Run the following to get the commands to drop all tables and their constraints:

```
begin
  for i in (select index_name from user_indexes where index_n
    execute immediate 'drop␣index␣' || i.index_name;
  end loop;

  for i in (select trigger_name from user_triggers) loop
    execute immediate 'drop␣trigger␣' || i.trigger_name;
  end loop;

  for i in (select view_name from user_views) loop
    execute immediate 'drop␣view␣' || i.view_name;
  end loop;
```

SQL

Operators

Operator	Description
=	Equality
!=,<>	Inequality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
IN	Equal to any value in a list of values
ANY/ SOME/ ALL	Compare a value to a list or subquery. It must be preceded by another operator such as =, >, <.j
NOT IN	Not equal to any value in a list of values
[NOT] BETWEEN n and m	Equivalent to [Not] >= n and <= y.
[NOT] EXISTS	Return true if subquery returns at least one row

Joins

- An **inner join** matches stuff in both tables:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b
```

- A **left (outer) join** matches everything in the left tables plus what matches in the right table:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b
```

- This **left (outer)** join matches everything that is in the left table and not in the right table:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b
```

- A **right (outer) join** matches everything in the right join plus what matches in the left table:

```
select a.id as id_a, a.color as color_a, b.id as id_b, b
```

Aliases

- You can alias long column names with
  select mylongname as name from contacts or just
  select mylongname name from contacts. The as keyword is optional.
  Full-text column names are supported by enclosing in “”. as can also format strings:
  select first_name || ' ' || last_name as ”Name” from employees;
  yields Alice, Bob and System.
- You can also create a table alias (using from employees e), but you CAN’T USE the as keyword.

Limits and Pagination

- The Oracle equivalent of filter is fetch n next rows only:
  select * from products order by list_price desc fetch next 5 rows only;.
- You may also use the fetch next n percent rows only:

```
select * from inventories order by quantity desc fetch n
```

- Filtering by for example a quantity, and you only want the first 10 "condition matches"? Use fetch n next rows with ties:

```
select * from inventories order by quantity desc fetch n
```

- Need Pagination? Use offset:

```
select * from products order by standard_cost desc offse
```

Dates and Intervals

- Want to extract a year from a date? Use extract:

```
select * from orders where status = 'Shipped' and extrac
```

- Want to get the current date? Use current_date:

```
select current_date from dual;
```

- The to_char function can convert dates (and timestamps) to chars:

```
select to_char(sysdate, 'YYYY-MM-DD') from dual;
```

- The to_date function can convert chars to dates:

```
select to_date('2021-01-12', 'YYYY-MM-DD') from dual;
```

- Alternatively, the date literal uses the YYYY-MM-DD format and does not require format specs:

```
select date '1969-04-20' from dual;
```

Expressions

- Only single quotes are supported.
- Comparisons are done with =, NOT ==.
- It also supports full expression evaluation:

```
select product_name as "Product Name", list_price - standard
```

- You can use () in where clauses to prioritize:

```
select * from orders where (
status = 'Canceled' or status = 'Pending' ) and customer
order by order_date;
```

- The in keyword is a useful tool for sub collections and subqueries:
  - select * from orders where salesman_id in (54, 55, 56) order by order_id;
  - select * from orders where salesman_id not in (54, 55, 56) order by order_id

Grouping and Ordering

- You can use functions like upper and dates when ordering.
- The group by keyword can be used to find unique data:

```
select status from orders group by status;
```

- By combining group by with count you can count the amount of unique data:

```
select status, count (*) from orders group by status;
```

- group by can also be used with the where keyword:

```
select name, count(*) as "Shipped␣Orders" from orders in
```

- where can NOT APPEAR AFTER group by; use the having keyword instead.
- The having keyword enables you to filter like with where, but after

Counting and Sums

- You can count the amount of rows with the count() function:

```
select count(*) from products
```

- The sum function can be used to calculate a total:

```
select sum(unit_price * quantity) from order_items;
```

- It can also be used to calculate a total per row (the group by order_id part is required; group by order_value does not work):

```
select order_id, sum(unit_price * quantity) as order_valu
```

Inserting

- It is a good idea to always specify the columns when inserting:

```
insert into discounts(discount_name, amount, start_date,
```

- You can also "insert from select" using insert into:

```
insert into sales(customer_id, product_id, order_date, t
```

- It's even possible to "create a table from select" using create table x as, basically coping its schema (where 1 = 0 skips copying the rows):

```
create table sales_2017 as select * from sales where 1 =
```

- Using insert all, it is possible to insert multiple rows at once (note the lack of commas between the into keywords. Here, the subquery is ignored/a placeholder.):

```
insert all into fruits (fruit_name, color) values ('Appl
```

Switches

- Using case it is possible to create if/else constructs:

```
select product_name, list_price, case category_id when 1
```

- case is also useful for conditional grouping:

```
select * from locations order by country_id, case countr
```

- case also evaluates to an expression, so you can use it for conditional updates:

```
update products set list_price = case when list_price <
```

Helper Functions

- You can extract substrings with substr:
  `select substr('Alex', 1, 1) from dual;`
- Stuff like `select upper('uwu') from dual` can come in handy.
- Using round it is possible to round numbers (returns 5.23):

```
select round(5.234234234234, 2) from dual;
```

- You can use replace to replace strings:

```
update accounts set phone = replace(phone, '+1-', '');
```

- You can use the floor, round and ceil functions to get rounded values.

Auto-Generated Primary Keys

- generated by default as identity is quite useful for auto-incrementing columns such as PKs:

```
create table persons ( person_id number generated by def
```

- generated always as identity is the same but does not allow setting it manually.

Modifying Columns

- You can use desc mytable to show the schema for a table.
- alter table can be used to add columns using add:

```
alter table persons add birthdate date not null;
```

- You can also add multiples at once (note that there is no column keyword):

```
alter table persons add ( phone varchar2(20), email varch
```

- modify can change the column type (note that there is no column keyword):

```
alter table persons modify birthdate date null;
```

- drop column can be used to remove a column

```
alter table persons drop column birthdate;
```

Virtual Columns

- You can create virtual columns in regular tables without using views with `alter table x add ... as` (note the required `(` after the `as` keyword):

```
alter table parts add (capacity_description as ( case wh
```

- The size of a varchar2 is adjustable afterwards (note that this checks if any current varchar2s are larger than the new size and fails if they are.):

```
alter table persons modify first_name varchar2(255);
```

Modifying Tables

- You can drop a table with drop table:

```
drop table people;
```

- Appending purge clears the recycle bin; appending cascade constraints drop all related constraints.

- You can clear a table using truncate table:

```
truncate table customers_copy;
```

- The same limitations as with drop table concerning constraints apply, so appending cascade (WITHOUT constraints) drops all related ones.

- You can clear the recycle bin with:

```
purge recyclebin;
```

Constraints

- It is possible to add constraints (any constraints, a primary key in this example) after creating a table with add constraint:

```
alter table purchase_orders add constraint purchase_orde
```

- You may remove a constraint with drop constraint:

```
alter table purchase_orders drop constraint purchase_ord
```

- Instead of removing it, you can also use disable constraint:

```
alter table purchase_orders disable constraint purchase_
```

- And re-enable it with enable constraint:

```
alter table purchase_orders enable constraint purchase_o
```

- You can also add foreign key constraints:

```
alter table suppliers add constraint suppliers_supplier_g
```

Types

- You can create a number within a range: number(1,0).
- The number type is used for all types of numbers by specifying precision and scale: number(6) (or number(6,0)) is a signed integer fitting 6 digits, number(6,2) is a float with two digits precision. The DB doesn't just cut of numbers, it rounds them.
- The float type can be emulated by the number type, i.e. float(2) is equal to number(38,2). The argument is in bits instead of digits though.
- The lengthdb function can be used to get the length of field in bytes.
- The char type has a fixed length: name char(10) or name char(10 bytes), meaning that a char always takes up the amount of bytes set. nchar is the same but UTF-8 or UTF-16 any doesn't take bytes.
- The varchar2 type also takes an argument for the length in bytes, which in ASCII corresponds to the amount of characters. nvarchar2 is

Views

- You can create a view with `create view x as select ...` :

  ```
  create view employees_years_of_service as select employe
  ```

- If used with `create or replace view`, upserts are possible.

- By appending with `read only`, you can prevent data modifications:

  ```
  create or replace view employees_years_of_service as sel
  ```

- `drop view x` removes the view.

- Deletions and updates on views are usually fine, but inserts can often be not that useful due to fields being excluded from the view; see instead of triggers later on for a solution;

- Subqueries can be used in selects:

  ```
  select * from ( select * from products) where list_price
  ```

Indexes

- You can create an index with create index:

```
create index members_last_name on members(last_name);
```

- You can also create an index spanning multiple columns:

```
create index members_full_name on members(first_name, las
```

- You can drop an index with drop index:

```
drop index members_full_name;
```

PL/SQL

Block Structure

- Block structure:

```
declare
-- declarations
begin
-- your logic
exception
-- exception handling
end;
```

- The most simple example is as follows:

```
begin
    dbms_output.put_line('Hello␣World!');
end;
```

- Use put_line from the dmbs_output package to print to stdout.

Variables

- PL/SQL extends SQL by adding a boolean type (which can have the values true, false and null).
- Variables need not be given a value at declaration if they are nullable:

```
declare
    total_sales number(15,2);
    credit_limit number(10,0);
    contact_name varchar2(255);
begin
    null;
end;
```

- You can use default as an alternative to the := operator when assigning variables in the declaration section. DO NOT use = when assignment, even re-assignment also uses :=.

Fetching Data

- Use select … into to fetch data into variables; %TYPE infers the type of a column:

```
declare
    customer_name customers.name%TYPE;
    customer_credit_limit customers.credit_limit%TYPE;
begin
    select
        name, credit_limit
    into
        customer_name, customer_credit_limit
    from customers where customer_id = 38;

    dbms_output.put_line(customer_name || ': ' || custom
end;
```

Branches and Expressions

- if … then … end if can be used for branching:

```
declare
    sales number := 20000;
begin
    if sales > 10000 then
        dbms_output.put_line('Lots␣of␣sales!');
    end if;
end;
```

- Inline expressions are also supported:

```
large_sales := sales > 10000
```

- Booleans need not be compared with my_bool = true, a simple if my_bool then is fine.

- elseif … then is NOT valid syntax; elsif … then is valid syntax.

Switches

- You may use the case keyword for switch cases:

```
declare
    grade char(1);
    message varchar2(255);
begin
    grade := 'A';

    case grade
        when 'A' then
            message := 'Excellent';
        when 'B' then
            message := 'Great';
        when 'C' then
            message := 'Good';
        when 'D' then
```

Labels and Goto

- A label/goto equivalent is also available:

```
begin
    goto do_work;
    goto goodbye;

    <<do_work>>
    dbms_output.put_line('mawahaha');

    <<goodbye>>
    dbms_output.put_line('Goodbye!');
end;
```

Loops

- The equivalent of the while loop is the loop. exit/continue prevents an infinite loop:

```
declare
    i number := 0;
begin
    loop
        i := i + 1;

        dbms_output.put_line('Iterator: ' || i);

        if i >= 10 then
            exit;
        end if;
    end loop;
```

Types and Objects

- You can also use %ROWTYPE to infer the type of a row and select an entire row at once:

```
declare
    customer customers%ROWTYPE;
begin
    select * into customer from customers where customer

    dbms_output.put_line(customer.name || '/' || custome
end;
```

- It is also possible to use OOP-style object/row creation thanks to %ROWTYPE:

```
declare
    person persons%ROWTYPE;
```

Exceptions

- You can create custom exceptions:

```
declare
    e_credit_too_high exception;
    pragma exception_init(e_credit_too_high, -20001);
begin
    if 10000 > 1000 then
        raise e_credit_too_high;
    end if;
end;
```

- If you want to raise a custom exception, use raise_application_error:

```
declare
    e_credit_too_high exception;
    pragma exception_init(e_credit_too_high, -20001);
begin
```

Cursors

- Using cursors, you can procedurally process data:

```
declare
    cursor sales_cursor is select * from sales;
    sales_record sales_cursor%ROWTYPE;
begin
    update customers set credit_limit = 0;

    open sales_cursor;

    loop
        fetch sales_cursor into sales_record;
        exit when sales_cursor%NOTFOUND;

        update
            customers
```

Locks

- The DB can also lock fields for safe multiple access:

```
declare
    cursor customers_cursor is select * from customers fo
begin
    for customer_record in customers_cursor loop
        update customers set credit_limit = 0 where custo
    end loop;
end;
```

Procedures

- You can create procedures, which are comparable to functions:

```
create or replace procedure
    print_contact(customer_id_arg number)
is
    contact_record contacts%rowtype;
begin
    select * into contact_record from contacts where cust

    dbms_output.put_line(contact_record.first_name || '␣
end;
```

- These procedures can then be executed:

```
begin
    print_contact(50);
end;
```

Functions

- Functions are similar, but require returning a value:

```
create or replace function
    get_total_sales_for_year(year_arg integer)
return number
is
    total_sales number := 0;
begin
    select sum(unit_price * quantity) into total_sales
    from order_items
    inner join orders using (order_id)
    where status = 'Shipped'
    group by extract(year from order_date)
    having extract(year from order_date) = year_arg;

    return total_sales;
```

Packages

- Packages can be used to group function “interfaces” and variables:

```
create or replace package order_management
as
    shipped_status constant varchar(10) := 'Shipped';
    pending_status constant varchar(10) := 'Pending';
    cancelled_status constant varchar(10) := 'Canceled';

    function get_total_transactions return number;
end order_management;
```

- You can now access the variables in the package with .:

```
begin
    dbms_output.put_line(order_management.shipped_status
end;
```

Triggers

- Triggers follow a similar structure as procedures:

```
declare
-- declarations
begin
-- your logic
exception
-- exception handling
end;
```

- Using triggers, you can for example create a manual log after operations with after update or delete on ... :

```
create or replace trigger customers_audit_trigger
    after update or delete
    on customers
    for each row
```

Maps

- Maps are also possible in PL/SQL using table of:

```
declare
    type country_capitals_type
        is table of varchar2(100)
        index by varchar2(50);

    country_capitals country_capitals_type;
begin
    country_capitals('China') := 'Beijing';
    country_capitals('EU') := 'Brussels';
    country_capitals('USA') := 'Washington';
end;
```

- You can use mymap.first and mymap.next to iterate:

```
declare
```

Arrays

- Using varray, it is also possible to create arrays:

```
declare
    type names_type is varray(255) of varchar2(20) not n

    names names_type := names_type('Albert', 'Jonathan',
begin
    dbms_output.put_line('Length␣before␣append:␣' || nam

    names.extend;

    names(names.last) := 'Alice';

    dbms_output.put_line('Length␣after␣append:␣' || name

    names.trim;
```