DB1 Notes

Tutorial

https://www.oracletutorial.com/oracle-basics/

Clean Start

```
select 'drop table ', table_name, 'cascade constraints;' from
user_tables;
```

Now paste the output into SQL Developer

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 =, >, <.
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
IS [NOT] NULL	NULL test

Quirks

Only single quotes are supported.

Stuff like `select upper('uwu') from dual` can come in handy.

Multiple order by statements? First ordered by first statement, then "sub-ordered" by the second (last name the same -> now first name is evaluated).

Want to have nulls first when ordering? Use `nulls first` or `nulls last` as the suffix.

You can use functions like `upper` and dates when ordering.

Removal of duplicates is done with `select distinct`. When multiple columns are being selected, use only one `distinct` keyword at the start. Multiple nulls are filtered (Null = Null).

`... like '%Asus%'` is basically a full-text search.

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:

Name
Alice
Bob
System

It also supports full expression evaluation:

```
select product_name as "Product Name", list_price - standard_cost
as "Gross Profit" from products order by "Gross Profit"
```

You can also create a table alias (`from employees e`), but you CAN'T USE the `as` keyword.

The Oracle equivalent of `filter` is `fetch n next rows only`: `select * from products order by list_price desc fetch next 5 rows only`.

Filtering by for example a quantity, and you only want the first 10 "condition matches"? Use `fext n next rows with ties`: `select * from inventories order by quantity desc fetch next 5 rows with ties;`.

You may also use the `fetch next n percent rows only`: `select * from inventories order by quantity desc fetch next 10 percent rows only;`

Pagination? Use offset: `select * from products order by standard_cost desc offset 10 rows fetch next 10 rows only;`.

Comparisons are done with `=`.

Want to extract a year from a date? Use `extract`: `select * from orders where status = 'Shipped' and extract(year from order_date) = 2017 order by order_date desc fetch next 1 rows with ties;`.

You can use `()` in where clauses to prioritize: `select * from orders where ( status = 'Canceled' or status = 'Pending' ) and customer_id = 44 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;` (you can use `not`)
- `select * from employees where employee_id in ( select distinct salesman_id from orders where status = 'Canceled' ) order by first_name;` (you can of course also use `not`)

Between can also be used for dates: `select * from orders where order_date between date '2016-12-01' and date '2016-12-31'`.

Some examples of `like` (you can use `not` for all of them):

- `select * from contacts where last_name like 'St%'`
- `select * from contacts where last_name like '%St'`
- `select * from contacts where last_name like '%St%'`
- `select * from contacts where last_name like 'Po_tinger'` (matches any one character)
- `select * from contacts where lower(last_name) like 'st%'`
- `select * from contacts where upper(last_name) like 'st%'`
- `select * from discounts where discount_message like '%%%'` (returns everything)
- `select * from discounts where discount_message like '%%%' escape '!'` (returns everything that includes the string '%')

You can compare against null with `is null` (`= NULL` does not work). You can negate with `not`.

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.color as color_b from palette_a a inner join palette_b b on a.color = b.color;`.

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.color as color_b from palette_a a left join palette_b b on a.color = b.color`

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.color as color_b from palette_a a left join palette_b b on a.color = b.color where b.id is null`.

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.color as color_b from palette_a a right join palette_b b on a.color = b.color;`

This right (outer) join matches everything that is in the right table and not in the left table: `select a.id as id_a, a.color as color_a, b.id as`

`id_b, b.color as color_b from palette_a a right join palette_b b on a.color = b.color where a.id is null;`

A full (outer) join merges both tables: `select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_a a full join palette_b b on a.color = b.color;`

This full (outer) join merges both tables and removes those rows which are in both: `select a.id as id_a, a.color as color_a, b.id as id_b, b.color as color_b from palette_a a full join palette_b b on a.color = b.color where a.id is null or b.id is null`

In addition to the `on` keyword you can also use the `using` keyword if the PK and FK are the same: `select * from orders inner join order_items using(order_id)`. You can also use multiple `on` or `using` statements: `select * from orders inner join order_items using(order_id) inner join customers using(customer_id)`. If you use the `on` keyword, use `and` for multiples!

You can also create the Cartesian product: `select * from products cross join warehouse;`.

It is also possible to do a self join: `select (w.first_name || ' ' || w.last_name) "Worker", (m.first_name || ' ' || m.last_name) "Manager", w.job_title from employees w left join employees m on w.employee_id = m.manager_id`.

You can count the amount of rows with the count() function: `select count(*) from products`.

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 inner join customers using(customer_id) where status = 'Shipped' group by name order by "Shipped Orders" desc`. `where` can NOT APPEAR AFTER `group by`; use the `having` keyword instead.

The `having` keyword enables you to filter like with `where`, but after the `group by` keyword like so: `select status from orders where extract(year from order_date) > '2016' group by status having status like '%d'`.

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: `select order_id, sum(unit_price * quantity) as order_value from order_items group`

`by order_id` (the `group by order_id` part is required; `group by order_value` does not work).

What is the difference between `join` and `union`? `join` merges horizontally (there are more columns than before, maybe also more rows), `union` merges vertically (there are more rows than before, but the column count stays the same).

`union` is similar to `T1 | T2` in TypeScript: `select first_name, last_name, email, 'contact' as role from contacts union select first_name, last_name, email, 'employee' as role from employees order by role` removes duplicates. Note that we have to use `select` two times.

`union all` is similar to `T1 & T2` in TypeScript: `select last_name from contacts union all select last_name from employees;` keeps duplicates.

Wish to find the difference between two tables? Use `intersect`: `select last_name from contacts intersect select last_name from employees;`.

Wish to have subtracted one table from another table? Use `minus`: `select last_name from contacts minus select last_name from employees;`.

It is a good idea to always specify the columns when inserting: `insert into discounts(discount_name, amount, start_date, expired_date) values ('Summer Promotion', 9.5, date '2017-05-01', date '2017-08-31')`.

Want to get the current date? Use `current_date`: `select current_date from dual;`.

You can also "insert from select" using `insert into`: `insert into sales(customer_id, product_id, order_date, total) select customer_id, product_id, order_date, sum(quantity * unit_price) amount from orders inner join order_items using(order_id) where status = 'Shipped' group by customer_id, product_id, order_date`.

It's even possible to "create a table from select" using `create table x as`, basically coping its schema: `create table sales_2017 as select * from sales where 1 = 0;`. `where 1 = 0` skips copying the rows.

Using `insert all`, it is possible to insert multiple rows at once: `insert all into fruits (fruit_name, color) values ('Apple', 'Red') into fruits (fruit_name, color) values ('Orange', 'Orange') into fruits (fruit_name, color) values ('Banana', 'Yellow') select 1 from dual`. Not the lack of commas between the `into` keywords. Here, the subquery is ignored/a placeholder.

You can also use conditions based on the subquery: `insert all when amount < 10000 then into small_orders when amount >= 10000 then into big_orders select order_id, customer_id, (quantity * unit_price) amount from orders inner join order_items using (order_id)`.

`insert first` is the equivalent of a switch case.

Using `case` it is possible to create `if/else` constructs: `select product_name, list_price, case category_id when 1 then round(list_price * 0.05, 2) when 2 then round(list_price * 0.1, 2) else round(list_price * 0.2, 2) end discount from products`.

`case` is also useful for conditional grouping: `select * from locations order by country_id, case country_id when 'US' then state else city end;`.

`case` also evaluates to an expression, so you can use it for conditional updates: `update products set list_price = case when list_price < 20 then 30 else 50 end where list_price < 50;`.

Using `round` it is possible to round numbers: `select round(5.234234234234, 2) from dual;` returns `5.23`.

`generated by default as identity` is quite useful for auto-incrementing columns such as PKs: `create table persons ( person_id number generated by default as identity, first_name varchar2(50) not null, last_name varchar2(50), primary key(person_id) )`. `generated always as identity` is the same but does not allow setting it manually.

`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: `alter table persons add ( phone varchar2(20), email varchar2(100) )`. Notice that there is not `column` keyword.

You can use `desc mytable` to show the schema for a table.

`modify` can change the column type: `alter table persons modify birthdate date null`. Notice that there is not `column` keyword.

`drop column` can be used to remove a column: `alter table persons drop column birthdate`.

`rename column` can be used to rename a column: `alter table persons rename column first_name to forename`.

`rename to` can be used to rename a table: `alter table persons rename to people`. `rename promotions to promotions_two` is an alternative syntax.

You can create virtual columns in regular tables without using views with `alter table x add ... as`: `alter table parts add (capacity_description as ( case when capacity <= 8 then 'Small' when capacity > 8 then 'Large' end ))`. Note the required `(` after the `as` keyword.

The size of a `varchar2` is adjustable afterwards: `alter table persons modify first_name varchar2(255)`. Note that this checks if any current `varchar2`s are larger than the new size and fails if they are.

You can use `replace` to replace strings: `update accounts set phone = replace(phone, '+1-', '')`.

Using the `generated always as` keyword with `modify` allows you to update virtual columns: `alter table accounts modify full_name varchar2(52) generated always as (last_name || ', ' || first_name)`

You can use the `default` keyword to set a default value: `alter table accounts add status number(1,0) default 1 not null`.

A more efficient logical version of `drop column` is `set unused column`: `alter table suppliers set unused column fax;`. You can now drop it using `alter table suppliers drop unused columns`.

If you want to physically drop a column, use `drop`: `alter table suppliers drop (email, phone)`.

You can drop a table with `drop table`: `drop table people`. Appending `purge` clears the recycle bin; `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 create a number within a range: `number(1,0)`.

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_orders_order_id_pk primary key(order_id);`.

You may remove a constraint with `drop constraint`: `alter table purchase_orders drop constraint purchase_orders_order_id_pk;`.

Instead of removing it, you can also use `disable constraint`: `alter table purchase_orders disable constraint purchase_orders_order_id_pk;` and re-enable it with `enable constraint`: ‘alter table purchase_orders enable constraint purchase_orders_order_id_pk;

You can also add foreign key constraints: `alter table suppliers add constraint suppliers_supplier_groups_fk foreign key(group_id) references supplier_groups(group_id);`

Using a check constraint, arbitrary expressions can be evaluated: `alter table parts add constraint check_buy_price_positive check(buy_price > 0);`.

A unique constraint prevents unwanted duplicates: `alter table clients add constraint unique_clients_phone unique(phone);`.

With a not null constraint, fuzzy logic can be avoided; it is however best to define nullable fields at schema creation, as the syntax differs from the `add constraint`/`drop constraint` logic above: `alter table clients modify (`

`phone not null );`. You can remove them by modifying it to null explicitly: `alter table clients modify ( phone null );`.

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 the same but UTF-8 or UTF-16 and doesn't take bytes.

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`.

You can get the current date with sysdate: `select localtimestamp from dual`. You can get the current datetime with localtimestamp: `select localtimestamp from dual`. The current time zone is available with sessiontimezone: `select sessiontimezone from dual` (yields Europe/Berlin).

The timestamp literal uses the `YYYY-MM-DD HH24:MI:SS.FF` format: `select timestamp '1969-04-20 00:00:00.00' from dual;`. You may also append the timezone: `select timestamp '1969-04-20 00:00:00.00 Europe/Berlin' from dual;`, but keep in mind that `timestamp with time zone` is the column type in this case.

The interval literal can be used to create intervals: `select interval '9' day from dual`, `select interval '9' month from dual`, `select interval '9-2' year to month from dual` or `select interval '09:08:6.75' hour to second(2) from dual`.

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

Using the `months_between` function, the count of months between two dates can be computed.

You can create a view with `create view x as select ...`: `create view employees_years_of_service as select employee_id, first_name || '`

`' || last_name as full_name, floor(months_between(current_date, hire_date) / 12) as years_of_service from employees;`. 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 select employee_id, first_name || ' ' || last_name as full_name, floor(months_between(current_date, hire_date) / 12) as years_of_service from employees with read only;`. `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.

Subqueries can be used in `selects`: `select * from ( select * from products) where list_price < 100;`.

They can also be used in `updates`: `update ( select list_price from products ) set list_price = list_price * 1.5`.