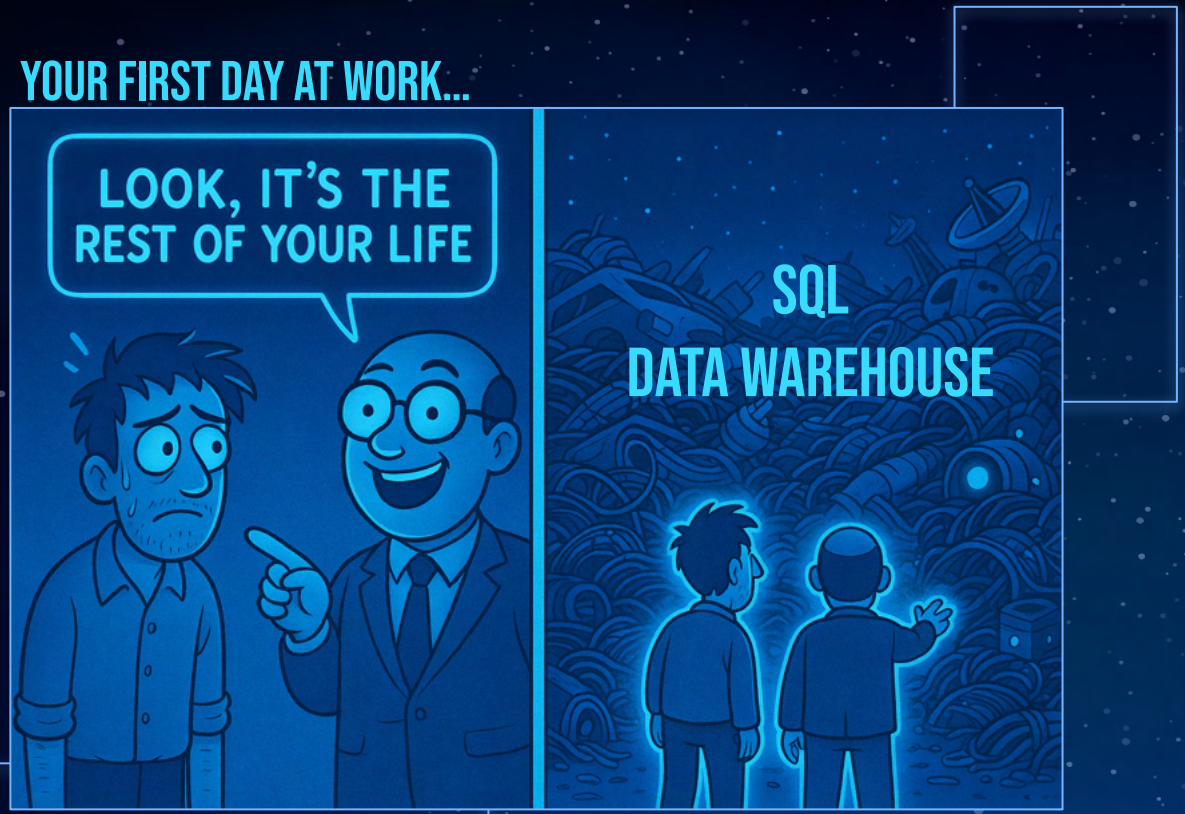
Language Integrated Query For Kotlin

Alexander Ioffe
github.com/ExoQuery

Table of contents

Table of contents


YOUR FIRST DAY AT WORK...
LOOK, IT'S THE REST OF YOUR LIFE
SQL
DATA WAREHOUSE

My Story

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) +
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  CASE
    WHEN code = 'EV'
    THEN cast(account.number as char)
    ELSE concat(cast(account.number as char), alias
  ) END AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'eu'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    pc.code AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      on entry.alias = serviceClient.alias
        and entry.record_type = 'S'
        AND entry.market = 'eu'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
    JOIN PARTNERSHIP_CODES pc
      on partnership.ID = pc.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  concat(cast(account.number as char), alias)
    AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'ca'
    AND entry.record_type = 'M'
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) +
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  CASE
    WHEN code = 'EV'
    THEN cast(account.number as char)
    ELSE concat(cast(account.number as char), alias
  ) END AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'eu'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    pc.code AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      on entry.alias = serviceClient.alias
        and entry.record_type = 'S'
        AND entry.market = 'eu'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
    JOIN PARTNERSHIP_CODES pc
      on partnership.ID = pc.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  concat(cast(account.number as char), alias)
    AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'ca'
    AND entry.record_type = 'M'
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

Independent Parts?

```
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  CASE
    WHEN code = 'EV'
    THEN cast(account.number as char)
    ELSE concat(cast(account.number as char), alias
  ) END AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'eu'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    pc.code AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      on entry.alias = serviceClient.alias
        and entry.record_type = 'S'
        AND entry.market = 'eu'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
    JOIN PARTNERSHIP_CODES pc
      on partnership.ID = pc.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

Service Clients

Merchant Clients

CTEs?

```
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  CASE
    WHEN code = 'EV'
    THEN cast(account.number as char)
    ELSE concat(cast(account.number as char), alias
  ) END AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'eu'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    pc.code AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      on entry.alias = serviceClient.alias
        and entry.record_type = 'S'
        AND entry.market = 'eu'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
    JOIN PARTNERSHIP_CODES pc
      on partnership.ID = pc.partnership_fk
) client
     INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
WITH merchantClients AS (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M')
serviceClients AS (
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
)
clients AS (merchantClients UNION ALL serviceClients)
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) +
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM clients client
     INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

Unsustainable Replication

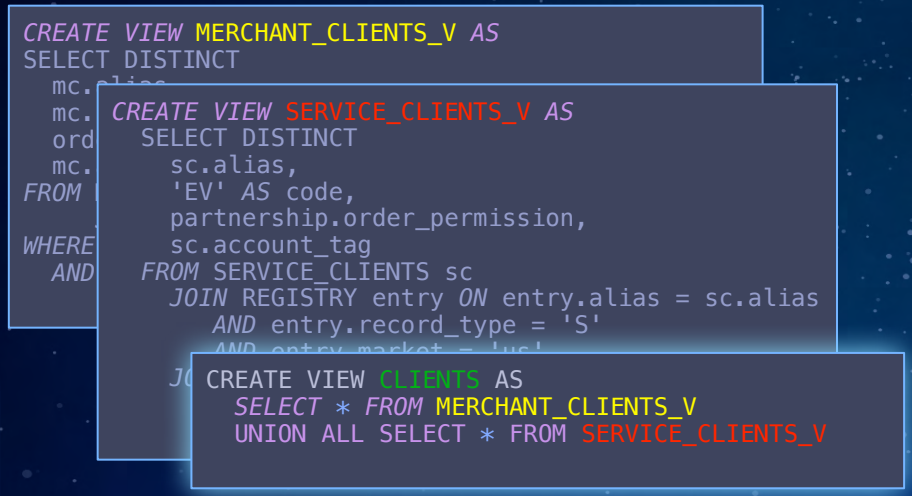
```sql
CREATE VIEW MERCHANT_CLIENTS_V AS
SELECT DISTINCT
  mc.alias,
  mc.code,
  order_permission,
  mc.account_tag
FROM MERCHANT_CLIENTS mc
    JOIN REGISTRY entry ON entry.alias = mc.alias
WHERE entry.market = 'us'
  AND entry.record_type = 'M'
```

Unsustainable Replication

```
CREATE VIEW MERCHANT_CLIENTS_V AS
SELECT DISTINCT
  mc.
  mc.
  ord
  mc.
FROM
WHERE
  AND
```

```
CREATE VIEW SERVICE_CLIENTS_V AS
  SELECT DISTINCT
    sc.alias,
    'EV' AS code,
    partnership.order_permission,
    sc.account_tag
  FROM SERVICE_CLIENTS sc
    JOIN REGISTRY entry ON entry.alias = sc.alias
      AND entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = sc.partnership_fk
```

Unsustainable Replication



"Client" Abstraction

Unsustainable Replication

```
CREATE VIEW MERCHANT_CLIENTS_V AS
SELECT DISTINCT
  mc.
  mc.
  ord
  mc.
FROM
WHERE
  AND
```

```
CREATE VIEW SERVICE_CLIENTS_V AS
  SELECT DISTINCT
    sc.alias,
    'EV' AS code,
    partnership.order_permission,
    sc.account_tag
  FROM SERVICE_CLIENTS sc
    JOIN REGISTRY entry ON entry.alias = sc.alias
      AND entry.record_type = 'S'
      AND entry.market = 'us'
    JO
```

```
CREATE VIEW CLIENTS AS
  SELECT * FROM MERCHANT_CLIENTS_V
  UNION ALL SELECT * FROM SERVICE_CLIENTS_V
```

```
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  CASE
    WHEN code = 'EV'
    THEN cast(account.number as char)
    ELSE concat(cast(account.number as char), alias
  ) END AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (CLIENTS) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

Unsustainable Replication

```
CREATE VIEW EU_SERVICE_CLIENTS AS
SELECT DISTINCT (... pc.code, ...)
FROM SERVICE_CLIENTS sc
     JOIN REGISTRY entry ON entry.alias = sc.alias
    AND entry.record_type = 'S'
    AND entry.market = 'eu'
     JOIN PARTNERSHIPS partnership
          ON partnership.id = sc.partnership_fk
     JOIN PARTNERSHIP_CODES pc
          ON partnership.ID = pc.partnership_fk
```

Unsustainable Replication

```
CREATE VIEW EU_SERVICE_CLIENTS AS
SELECT DISTINCT (... pc.code, ...)
FROM SERVICE_CLIENTS sc
     JOIN REGISTRY entry ON entry.alias = sc.alias
    AND entry.record_type = 'S'
    AND entry.market = 'eu'
     JOIN PARTNERSHIPS partnership
          ON partnership.id = sc.partnership_fk
     JOIN PARTNERSHIP_CODES pc
          ON partnership.ID = pc.partnership_fk
```

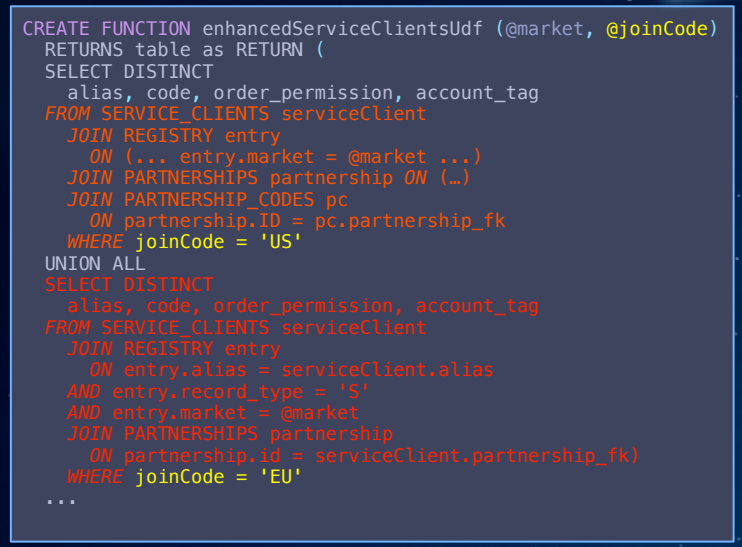
```
CREATE VIEW EU_CLIENT_ACCOUNTS AS
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  CASE
    WHEN code = 'EV'
    THEN cast(account.number as char)
    ELSE concat(cast(account.number as char), alias
  ) END AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (EU_CLIENTS) client
     INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
CREATE VIEW EU_CLIENTS AS
  SELECT * FROM MERCHANT_CLIENTS_V
  UNION ALL SELECT * FROM SERVICE_CLIENTS_V
```

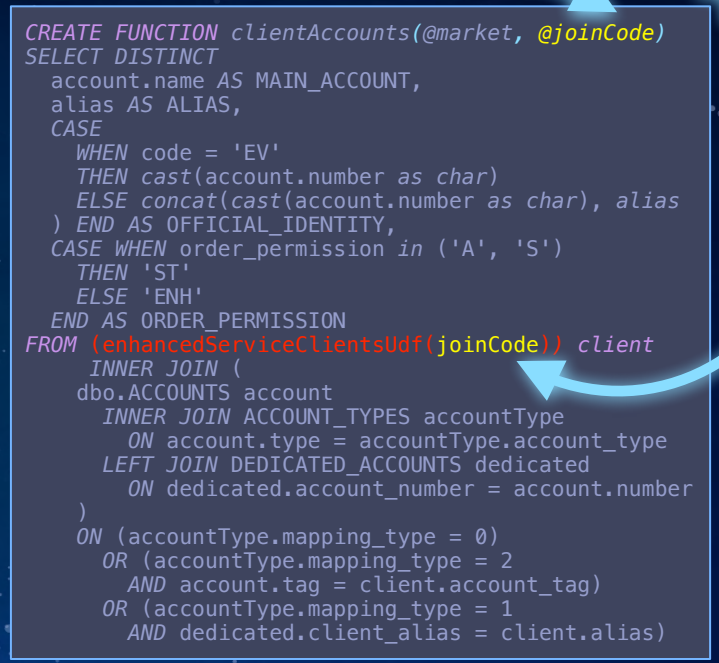
Unsustainable Repropagation

```
CREATE FUNCTION enhancedServiceClientsUdf (@market, @joinCode)
  RETURNS table as RETURN (
  SELECT DISTINCT
    alias, code, order_permission, account_tag
  FROM SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON (... entry.market = @market ...)
    JOIN PARTNERSHIPS partnership ON (…)
    JOIN PARTNERSHIP_CODES pc
      ON partnership.ID = pc.partnership_fk
    WHERE joinCode = 'US'
  UNION ALL
  SELECT DISTINCT
    alias, code, order_permission, account_tag
  FROM SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
    AND entry.record_type = 'S'
    AND entry.market = @market
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk)
    WHERE joinCode = 'EU'
  ...
```

Unsustainable Repropagation



```
CREATE FUNCTION enhancedServiceClientsUdf (@market, @joinCode)
  RETURNS table as RETURN (
  SELECT DISTINCT
    alias, code, order_permission, account_tag
  FROM SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON (... entry.market = @market ...)
    JOIN PARTNERSHIPS partnership ON (…)
    JOIN PARTNERSHIP_CODES pc
      ON partnership.ID = pc.partnership_fk
    WHERE joinCode = 'US'
  UNION ALL
  SELECT DISTINCT
    alias, code, order_permission, account_tag
  FROM SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
    AND entry.record_type = 'S'
    AND entry.market = @market
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk)
    WHERE joinCode = 'EU'
  ...
```



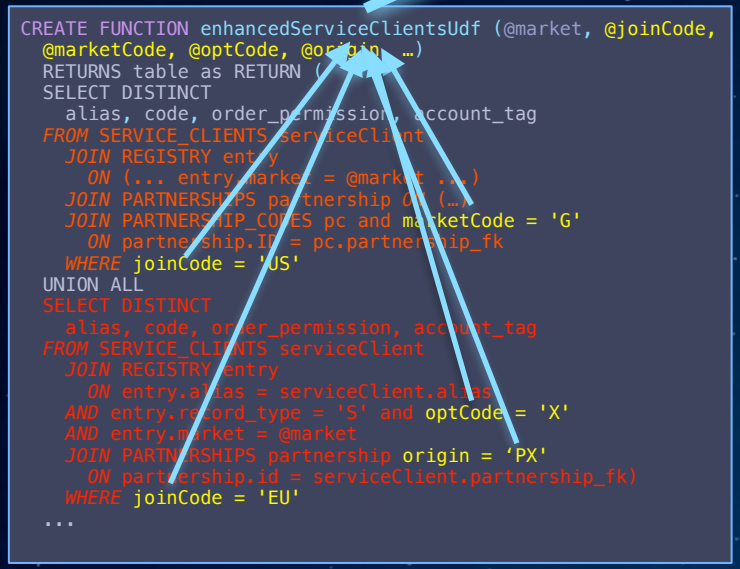
```
CREATE FUNCTION clientAccounts(@market, @joinCode)
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  CASE
    WHEN code = 'EV'
    THEN cast(account.number as char)
    ELSE concat(cast(account.number as char), alias
  ) END AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (enhancedServiceClientsUdf(joinCode)) client
     INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

Unsustainable Repropagation

```
CREATE FUNCTION enhancedServiceClientsUdf (@market, @joinCode,
  @marketCode, @optCode, @origin, …)
  RETURNS table as RETURN (
  SELECT DISTINCT
    alias, code, order_permission, account_tag
  FROM SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON (... entry.market = @market ...)
    JOIN PARTNERSHIPS partnership ON (…)
    JOIN PARTNERSHIP_CODES pc and marketCode = 'G'
      ON partnership.ID = pc.partnership_fk
    WHERE joinCode = 'US'
  UNION ALL
  SELECT DISTINCT
    alias, code, order_permission, account_tag
  FROM SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
    AND entry.record_type = 'S' and optCode = 'X'
    AND entry.market = @market
    JOIN PARTNERSHIPS partnership origin = 'PX'
      ON partnership.id = serviceClient.partnership_fk)
    WHERE joinCode = 'EU'
  ...
```

```
CREATE FUNCTION clientAccounts(@market, @joinCode)
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  CASE
    WHEN code = 'EV'
    THEN cast(account.number as char)
    ELSE concat(cast(account.number as char), alias
  ) END AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (enhancedServiceClientsUdf(joinCode)) client
     INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

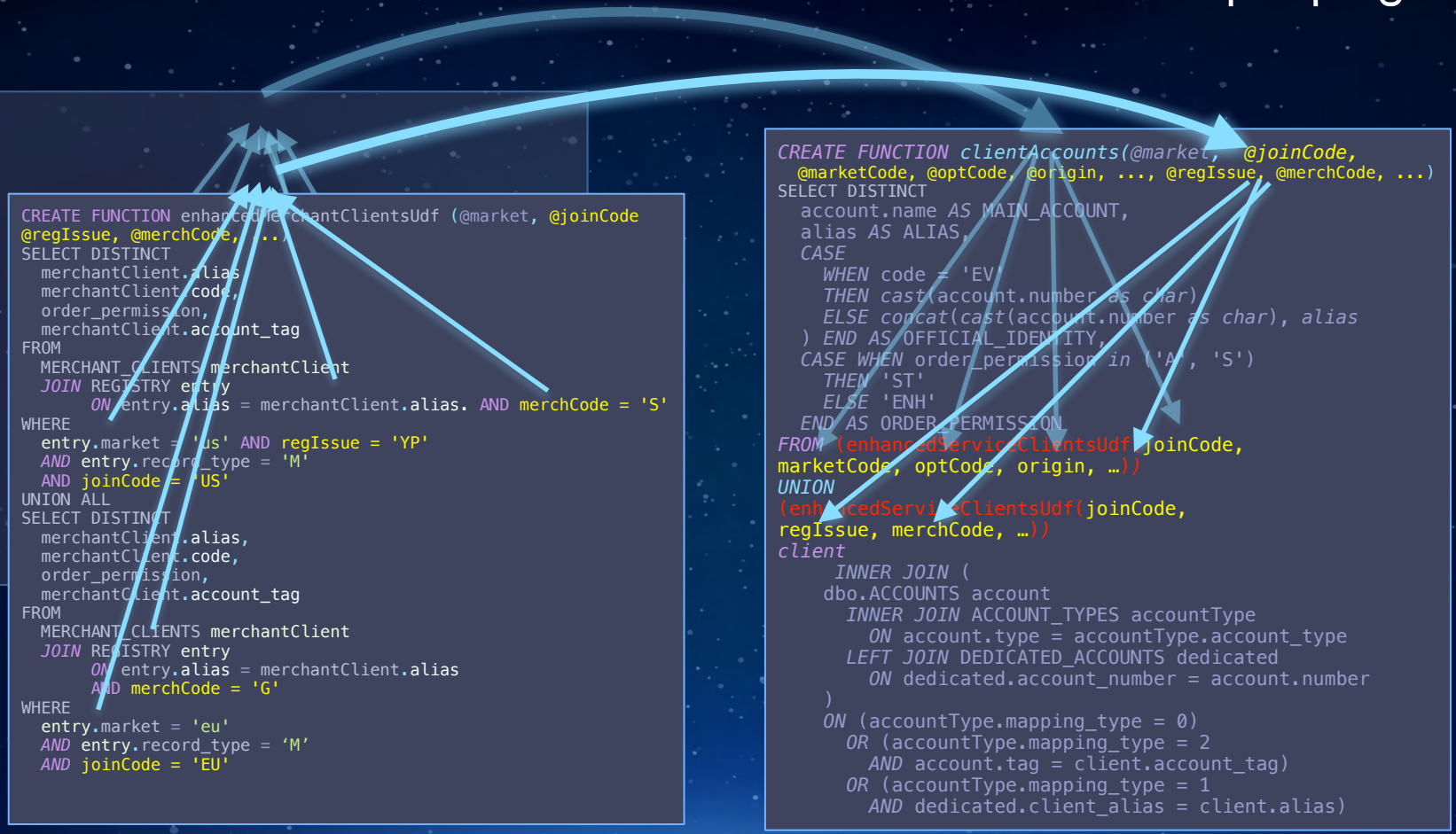
Unsustainable Repropagation



```
CREATE FUNCTION enhancedServiceClientsUdf (@market, @joinCode,
  @marketCode, @optCode, @origin, …)
  RETURNS table as RETURN (
  SELECT DISTINCT
    alias, code, order_permission, account_tag
  FROM SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON (... entry.market = @market ...)
    JOIN PARTNERSHIPS partnership ON (…)
    JOIN PARTNERSHIP_CODES pc and marketCode = 'G'
      ON partnership.ID = pc.partnership_fk
    WHERE joinCode = 'US'
  UNION ALL
  SELECT DISTINCT
    alias, code, order_permission, account_tag
  FROM SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
    AND entry.record_type = 'S' and optCode = 'X'
    AND entry.market = @market
    JOIN PARTNERSHIPS partnership origin = 'PX'
      ON partnership.id = serviceClient.partnership_fk)
    WHERE joinCode = 'EU'
  ...
```

```
CREATE FUNCTION clientAccounts(@market,  @joinCode,
  @marketCode, @optCode, @origin, …)
SELECT DISTINCT
  account.name AS MAIN_ACCOUNT,
  alias AS ALIAS,
  CASE
    WHEN code = 'EV'
    THEN cast(account.number as char)
    ELSE concat(cast(account.number as char), alias
  ) END AS OFFICIAL_IDENTITY,
  CASE WHEN order_permission in ('A', 'S')
    THEN 'ST'
    ELSE 'ENH'
  END AS ORDER_PERMISSION
FROM (enhancedServiceClientsUdf(joinCode,
marketCode, optCode, origin, …)) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

Unsustainable Repropagation



Unsustainable Repropagation

```
CREATE FUNCTION enhancedMerchantClientsUdf (@market, @joinCode
@regIssue, @merchCode, ...)
SELECT DISTINCT
  merchantClient.alias,
  merchantClient.code,
  order_permission,
  merchantClient.account_tag
FROM
  MERCHANT_CLIENTS merchantClient
  JOIN REGISTRY entry
       ON entry.alias = merchantClient.alias. AND merchCode = 'S'
WHERE
  entry.market = 'us' AND regIssue = 'YP'
  AND entry.record_type = 'M'
  AND joinCode = 'US'
UNION ALL
SELECT DISTINCT
  merchantClient.alias,
  merchantClient.code,
  order_permission,
  merchantClient.account_tag
FROM
  MERCHANT_CLIENTS merchantClient
  JOIN REGISTRY entry
       ON entry.alias = merchantClient.alias
       AND merchCode = 'G'
WHERE
  entry.market = 'eu'
  AND entry.record_type = 'M'
  AND joinCode = 'EU'
```

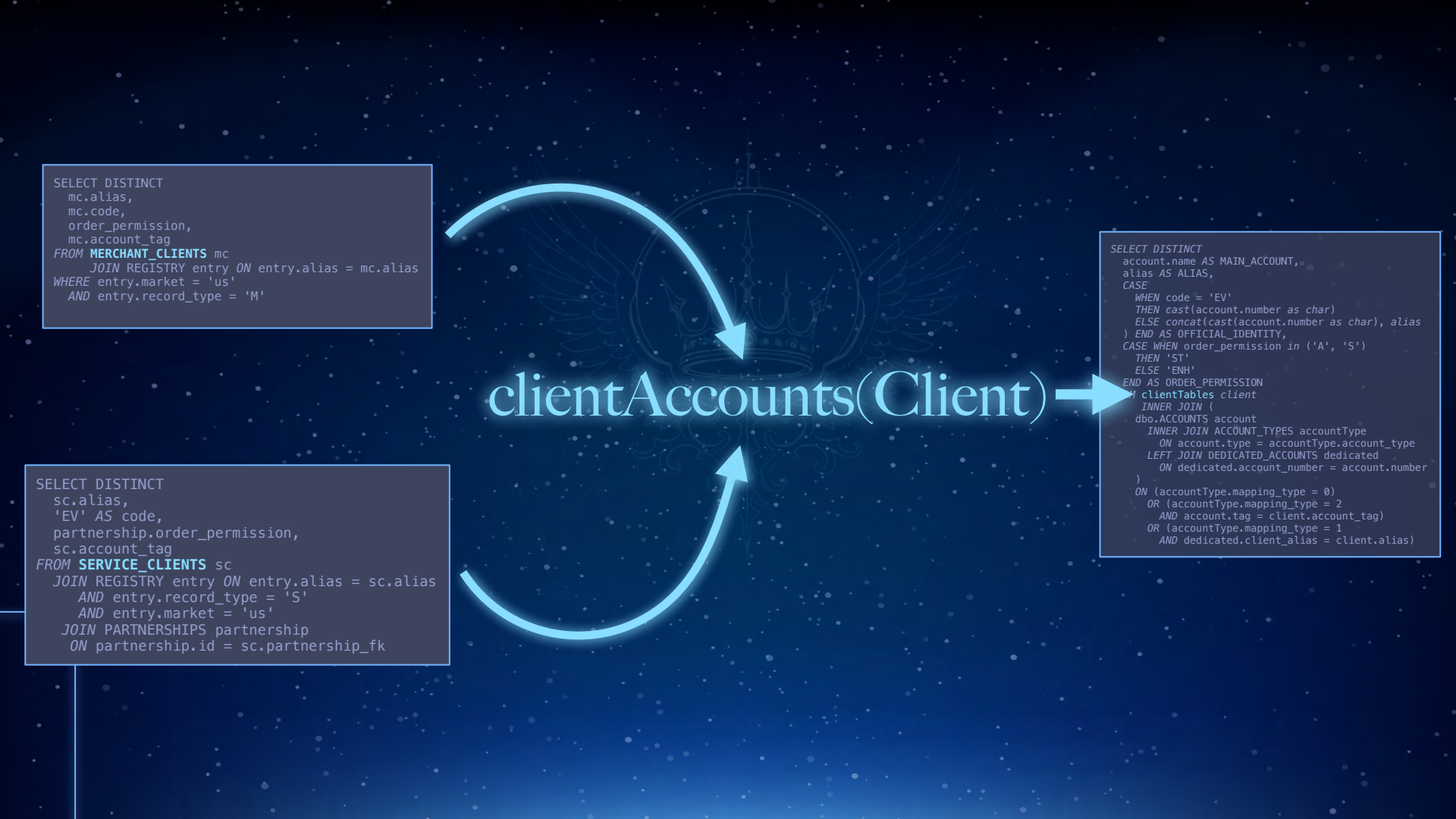
T - Intermediate Tables
C - Intermediate Categories
$(T/C)^2$ - Extra Unions Per Query

The API we need...

clientAccounts(Client): ClientAccount

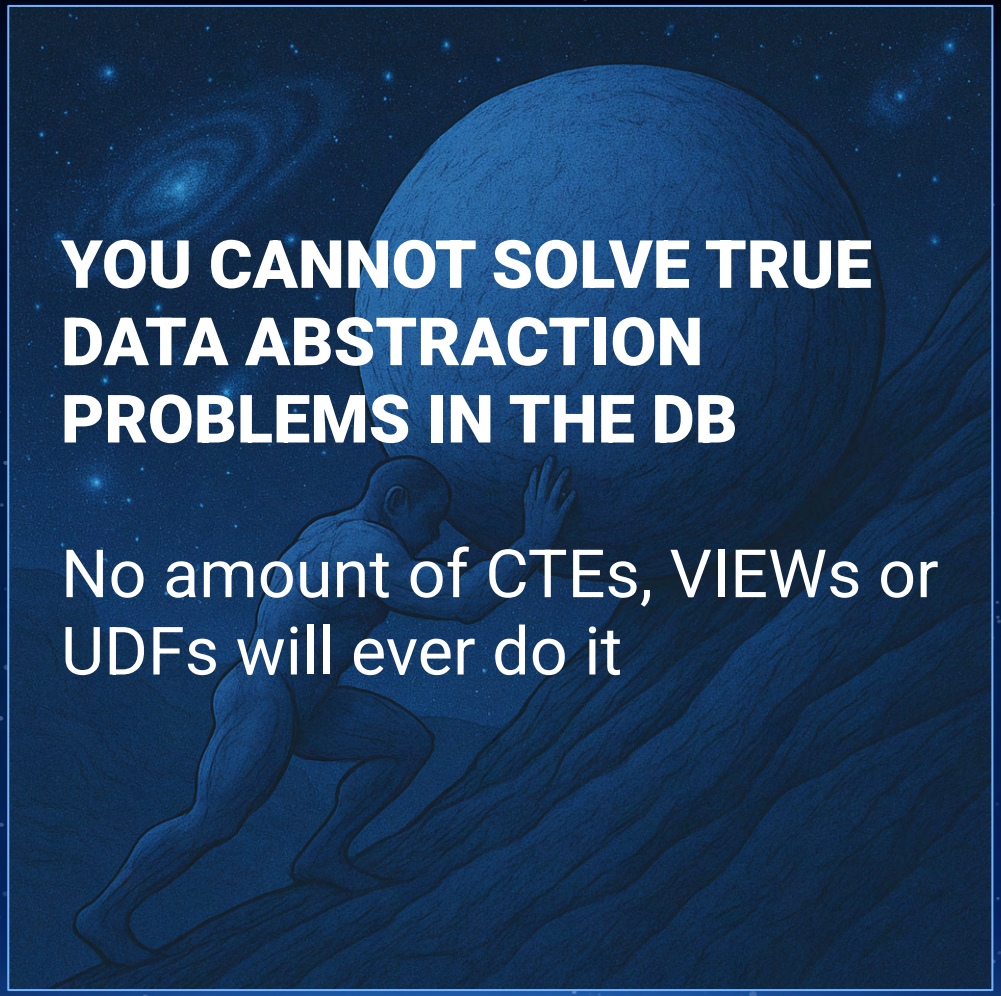
The API we need...

clientAccounts(List[Client]): List[ClientAccount]


SELECT DISTINCT
mc.alias,
mc.code,
order_permission,
mc.account_tag
FROM MERCHANT_CLIENTS mc
JOIN REGISTRY entry ON entry.alias = mc.alias
WHERE entry.market = 'us'
AND entry.record_type = 'M'
SELECT DISTINCT
sc.alias,
'EV' AS code,
partnership.order_permission,
sc.account_tag
FROM SERVICE_CLIENTS sc
JOIN REGISTRY entry ON entry.alias = sc.alias
AND entry.record_type = 'S'
AND entry.market = 'us'
JOIN PARTNERSHIPS partnership
ON partnership.id = sc.partnership_fk
clientAccounts(Client)
SELECT DISTINCT
account.name AS MAIN_ACCOUNT,
alias AS ALIAS,
CASE
WHEN code = 'EV'
THEN cast(account.number as char)
ELSE concat(cast(account.number as char), alias
) END AS OFFICIAL_IDENTITY,
CASE WHEN order_permission in ('A', 'S')
THEN 'ST'
ELSE 'ENH'
END AS ORDER_PERMISSION
FM clientTables client
INNER JOIN (
dbo.ACCOUNTS account
INNER JOIN ACCOUNT_TYPES accountType
ON account.type = accountType.account_type
LEFT JOIN DEDICATED_ACCOUNTS dedicated
ON dedicated.account_number = account.number
)
ON (accountType.mapping_type = 0)
OR (accountType.mapping_type = 2
AND account.tag = client.account_tag)
OR (accountType.mapping_type = 1
AND dedicated.client_alias = client.alias)

A polymorphic object!

No database can do this...


YOU CANNOT SOLVE TRUE DATA ABSTRACTION PROBLEMS IN THE DB
No amount of CTEs, VIEWs or UDFs will ever do it

clientAccounts(Client) →

No database can do this...

```
fun clientAccounts(Table<Client>): Table<ClientAccount>
```

... perhaps  **Kotlin** can?

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) +
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

...perhaps a **Kotlin** ecosystem library can?

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) +
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

~~Ktorm~~

Exposed

```
val merchantClientSubquery = MerchantClients
  .join(Registry, JoinType.INNER, onColumn = MerchantClients.alias, otherColumn = Registry.alias)
  .select(
    MerchantClients.alias,
    MerchantClients.code,
    MerchantClients.orderPermission,
    MerchantClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "M")
  }
  .withDistinct()
```

```
val merchantClientSubquery = MerchantClients
  .join(Registry, JoinType.INNER, onColumn = MerchantClients.alias, otherColumn = Registry.alias)
  .select(
    MerchantClients.alias,
    MerchantClients.code,
    MerchantClients.orderPermission,
    MerchantClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "M")
  }
  .withDistinct()
```

```
val serviceClientSubquery = ServiceClients
  .join(Registry, JoinType.INNER, onColumn = ServiceClients.alias, otherColumn = Registry.alias)
  .join(Partnerships, JoinType.INNER, onColumn = ServiceClients.partnershipFk, otherColumn = Partnerships.id)
  .select(
    ServiceClients.alias,
    stringLiteral("EV").alias("code"),
    Partnerships.orderPermission,
    ServiceClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "S")
  }
  .withDistinct()
```

```scala
val merchantClientSubquery = MerchantClients
  .join(Registry, JoinType.INNER, onColumn = MerchantClients.alias, otherColumn = Registry.alias)
  .select(
    MerchantClients.alias,
    MerchantClients.code,
    MerchantClients.orderPermission,
    MerchantClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "M")
  }
  .withDistinct()
```

```scala
val serviceClientSubquery = ServiceClients
  .join(Registry, JoinType.INNER, onColumn = ServiceClients.alias, otherColumn = Registry.alias)
  .join(Partnerships, JoinType.INNER, onColumn = ServiceClients.partnershipFk, otherColumn = Partnerships.id)
  .select(
    ServiceClients.alias,
    stringLiteral("EV").alias("code"),
    Partnerships.orderPermission,
    ServiceClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "S")
  }
  .withDistinct()
```

```scala
val unionQuery =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
```

```
val merchantClientSubquery = MerchantClients
  .join(Registry, JoinType.INNER, onColumn = MerchantClients.alias, otherColumn = Registry.alias)
  .select(
    MerchantClients.alias,
    MerchantClients.code,
    MerchantClients.orderPermission,
    MerchantClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "M")
  }
  .withDistinct()
```

```
val serviceClientSubquery = ServiceClients
  .join(Registry, JoinType.INNER, onColumn = ServiceClients.alias, otherColumn = Registry.alias)
  .join(Partnerships, JoinType.INNER, onColumn = ServiceClients.partnershipFk, otherColumn = Partnerships.id)
  .select(
    ServiceClients.alias,
    stringLiteral("EV").alias("code"),
    Partnerships.orderPermission,
    ServiceClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "S")
  }
  .withDistinct()
```

```
val unionQuery =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
val unionQuery =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
val unionQuery =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
fun clientAccounts(clients) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = clients.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        clients.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), clients.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq clients.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq clients.alias))
    }
```

Wait... what are the types?

```
val unionQuery =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
fun clientAccounts(clients) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = clients.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        clients.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), clients.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq clients.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq clients.alias))
    }
```

Wait... what are the types?

```
val unionQuery:QueryAlias =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
fun clientAccounts(clients: QueryAlias) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = clients.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        clients.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), clients.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq clients.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq clients.alias))
    }
```

Columns? From where?

```
val unionQuery:QueryAlias =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
object UnionSubquery : Table("client") {
  val alias = varchar("alias", 255)
  val code = varchar("code", 255)
  val orderPermission = varchar("order_permission", 1)
  val accountTag = varchar("account_tag", 4)
}
```

```
fun clientAccounts(clients) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = clients.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        clients.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), clients.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq clients.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq clients.alias))
    }
```

Columns? From where?

```
val unionQuery:QueryAlias =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
object UnionSubquery : Table("client") {
  val alias = varchar("alias", 255)
  val code = varchar("code", 255)
  val orderPermission = varchar("order_permission", 1)
  val accountTag = varchar("account_tag", 4)
}
```

```
fun clientAccounts(clients) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = clients.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        clients.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), clients.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq clients.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq clients.alias))
    }
```

Columns? From where?

```
val unionQuery:QueryAlias =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
object UnionSubquery : Table("client") {
  val alias = varchar("alias", 255)
  val code = varchar("code", 255)
  val orderPermission = varchar("order_permission", 1)
  val accountTag = varchar("account_tag", 4)
}
```

```
fun clientAccounts(clients) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = UnionSubquery.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        UnionSubquery.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), UnionSubquery.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq UnionSubquery.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq UnionSubquery.alias))
    }
```

Columns? From where?

```
val unionQuery:QueryAlias =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
object UnionSubquery : Table("client") {
  val alias = varchar("alias", 255)
  val code = varchar("code", 255)
  val orderPermission = varchar("order_permission", 1)
  val accountTag = varchar("account_tag", 4)
}
```

```
fun clientAccounts(clients) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = UnionSubquery.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        UnionSubquery.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), UnionSubquery.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq UnionSubquery.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq UnionSubquery.alias))
    }
```

```
val unionQuery:QueryAlias =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
object UnionSubquery : Table("client") {
  val alias = varchar("alias", 255)
  val code = varchar("code", 255)
  val orderPermission = varchar("order_permission", 1)
  val accountTag = varchar("account_tag", 4)
}
```

...but this is Hard Coded!

Can we our desired Query->Query function?

```
val unionQuery:QueryAlias =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
class UnionSubqueryBase(alias: String) : Table(alias) {
  val alias = varchar("alias", 255)
  val code = varchar("code", 255)
  val orderPermission = varchar("order_permission", 1)
  val accountTag = varchar("account_tag", 4)
}
```

```
fun clientAccounts(clients) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = UnionSubquery.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        UnionSubquery.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), UnionSubquery.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq UnionSubquery.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq UnionSubquery.alias))
    }
```

Wait... what are the types?

```
val unionQuery:QueryAlias =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
class UnionSubqueryBase(alias: String) : Table(alias) {
  val alias = varchar("alias", 255)
  val code = varchar("code", 255)
  val orderPermission = varchar("order_permission", 1)
  val accountTag = varchar("account_tag", 4)
}
```

```
fun clientAccounts(clients:QueryAlias) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = UnionSubquery.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        UnionSubquery.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), UnionSubquery.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq UnionSubquery.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq UnionSubquery.alias))
    }
```

Wait... what are the types?

```
val unionQuery:QueryAlias =
  merchantClientSubquery
  .unionAll(serviceClientSubquery)
  .alias("client")
```

```
class UnionSubqueryBase(alias: String) : Table(alias) {
  val alias = varchar("alias", 255)
  val code = varchar("code", 255)
  val orderPermission = varchar("order_permission", 1)
  val accountTag = varchar("account_tag", 4)
}
```

```
fun clientAccounts(clients:QueryAlias) = run {
  val UnionSubquery = UnionSubqueryBase(clients.alias)
  clients
    .join(accountQuery, JoinType.INNER, onColumn = UnionSubquery.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        UnionSubquery.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), UnionSubquery.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq UnionSubquery.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq UnionSubquery.alias))
    }
}
```

Did we really solve the problem?

```
fun clientAccounts(clients:QueryAlias) = run {
  val UnionSubquery = UnionSubqueryBase(clients.alias)
  clients
    .join(accountQuery, JoinType.INNER, onColumn = UnionSubquery.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        UnionSubquery.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), UnionSubquery.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq UnionSubquery.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq UnionSubquery.alias))
    }
}
```

Did we really solve the problem?

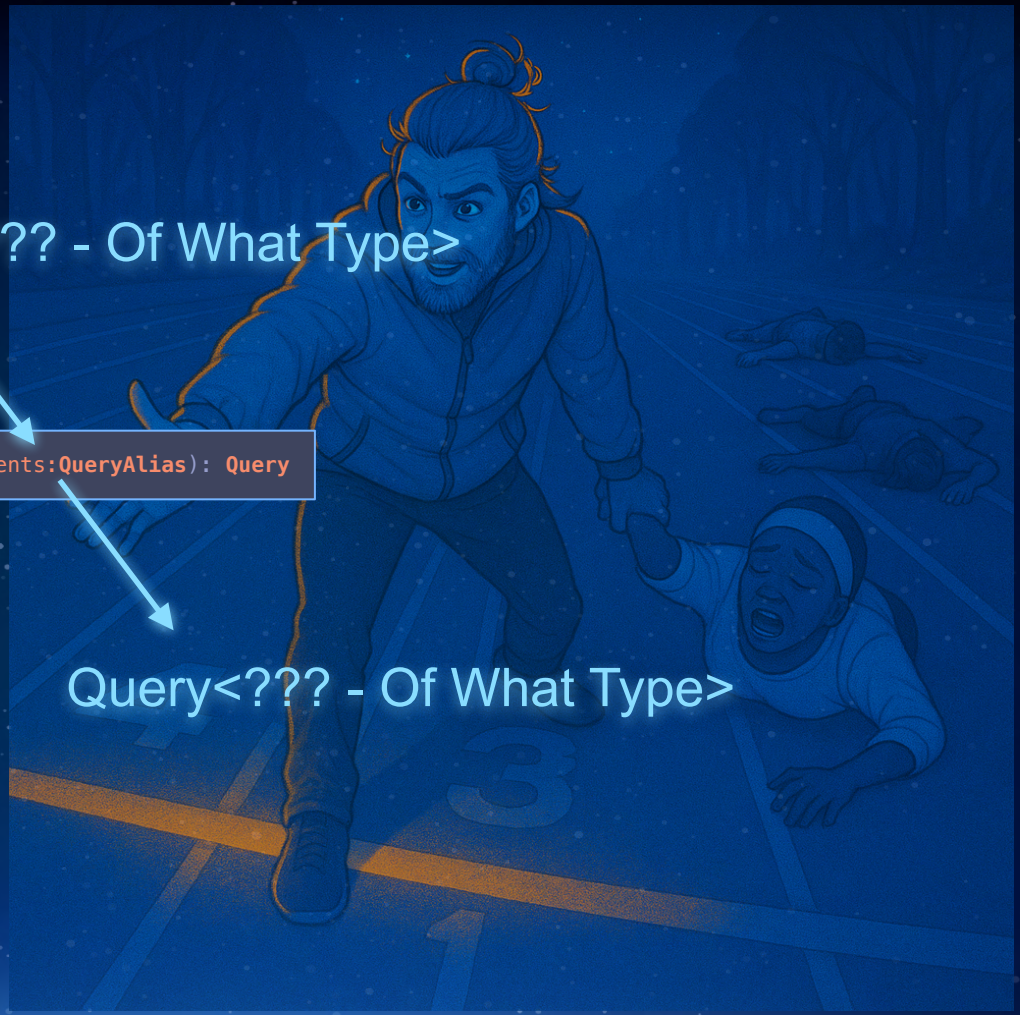
```
fun clientAccounts(clients:QueryAlias) = run {
  val UnionSubquery = UnionSubqueryBase(clients.alias)
  clients
    .join(accountQuery, JoinType.INNER, onColumn = UnionSubquery.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        UnionSubquery.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), UnionSubquery.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq UnionSubquery.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq UnionSubquery.alias))
    }
}
```

```
fun clientAccounts(clients:QueryAlias): Query
```

QueryAlias<??? - Of What Type>

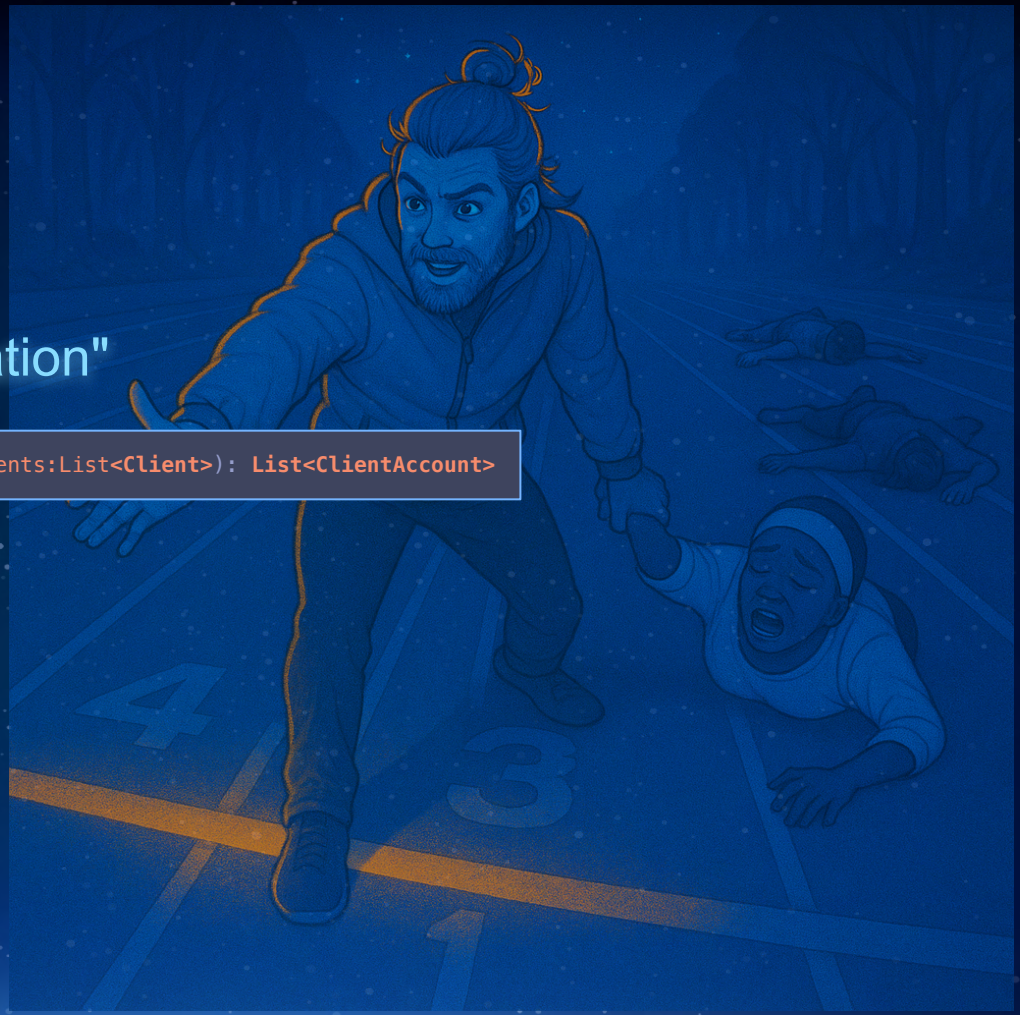
```
fun clientAccounts(clients:QueryAlias): Query
```

Query<??? - Of What Type>



The "Expectation"

```
fun clientAccounts(clients:List<Client>): List<ClientAccount>
```



```
fun clientAccounts(clients:QueryAlias): Query
```

≠

Just a small side-point?

```
fun clientAccounts(clients:QueryAlias) = run {
  val UnionSubquery = UnionSubqueryBase(clients.alias)
  clients
    .join(accountQuery, JoinType.INNER, onColumn = UnionSubquery.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        UnionSubquery.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), UnionSubquery.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq UnionSubquery.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq UnionSubquery.alias))
    }
}
```

```
Case().When(
  UnionSubquery.code eq stringLiteral("EV"),
  AccountsSubquery.number.castTo(VarCharColumnType(255))
).Else(
  concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), UnionSubquery.alias.substring(0, 2))
).alias("OFFICIAL_IDENTITY"),
```

```
Case().When(
  UnionSubquery.code:Column<Int> eq stringLiteral("EV"):Column<Int>,
  AccountsSubquery.number:Column<Int>.castTo(VarCharColumnType(255):Column<String>)
).Else(
  concat(AccountsSubquery.number:Column<Int>.castTo(VarCharColumnType(255)),
         UnionSubquery.alias.substring(0, 2):Column<String>
  )
).alias("OFFICIAL_IDENTITY"),
```

```
Case().When(
  UnionSubquery.code:Column<Int> eq stringLiteral("EV"):Column<Int>,
  AccountsSubquery.number:Column<Int>.castTo(VarCharColumnType(255):Column<String>)
).Else(
  concat(AccountsSubquery.number:Column<Int>.castTo(VarCharColumnType(255)),
         UnionSubquery.alias.substring(0, 2):Column<String>
  )
).alias("OFFICIAL_IDENTITY"),
```

```
Case().When(
  UnionSubquery.code:Column<Int> eq stringLiteral("EV"):Column<Int>,
  AccountsSubquery.number:Column<Int>.castTo(VarCharColumnType(255):Column<String>)
).Else(
  concat(AccountsSubquery.number:Column<Int>.castTo(VarCharColumnType(255)),
         UnionSubquery.alias.substring(0, 2):Column<String>
  )
).alias("OFFICIAL_IDENTITY"),
```

Consequences of Column<T> Types...

Consequences of Column<T> Types...

```
val merchantClientSubquery = MerchantClients
  .join(Registry, JoinType.INNER, onColumn = MerchantClients.alias, otherColumn = Registry.alias)
  .select(
    MerchantClients.alias,
    MerchantClients.code,
    MerchantClients.orderPermission,
    MerchantClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "M")
  }
  .withDistinct()
```

Consequences of Column<T> Types...

```
val merchantClientSubquery = MerchantClients
  .join(Registry, JoinType.INNER, onColumn = MerchantClients.alias, otherColumn = Registry.alias)
  .select(
    MerchantClients.alias,
    MerchantClients.code,
    MerchantClients.orderPermission,
    MerchantClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "M")
  }
  .wi
```

```
val serviceClientSubquery = ServiceClients
  .join(Registry, JoinType.INNER, onColumn = ServiceClients.alias, otherColumn = Registry.alias)
  .join(Partnerships, JoinType.INNER, onColumn = ServiceClients.partnershipFk, otherColumn = Partnerships.id)
  .select(
    ServiceClients.alias,
    stringLiteral("EV").alias("code"),
    Partnerships.orderPermission,
    ServiceClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "S")
  }
  .withDistinct()
```

Consequences of Column<T> Types...

```
val merchantClientSubquery = MerchantClients
  .join(Registry, JoinType.INNER, onColumn = MerchantClients.alias, otherColumn = Registry.alias)
  .select(
    MerchantClients.alias,
    MerchantClients.code,
    MerchantClients.orderPermission,
    MerchantClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "M")
  }
  .wi
```

```
val serviceClientSubquery = ServiceClients
  .join(Registry, JoinType.INNER, onColumn = ServiceClients.alias, otherColumn = Registry.alias)
  .join(Partnerships, JoinType.INNER, onColumn = ServiceClients.partnershipFk, otherColumn = Partnerships.id)
  .select(
    ServiceClients.alias,
    stringLiteral("EV").alias("code"),
    Partnerships.orderPermission,
    ServiceClients.accountTag
  )
  .where {
    (Registry.market eq "us") and (Registry.recordType eq "S")
  }
  .withDistinct()
```

```
fun clientAccounts(clients: QueryAlias) =
  clients
    .join(accountQuery, JoinType.INNER, onColumn = clients.alias, otherColumn = AccountsSubquery.alias)
    .select(
      AccountsSubquery.name,
      clients.alias,
      Case().When(
        clients.code eq stringLiteral("EV"),
        AccountsSubquery.number.castTo(VarCharColumnType(255))
      ).Else(
        concat(AccountsSubquery.number.castTo(VarCharColumnType(255)), clients.alias.substring(0, 2))
      ).alias("OFFICIAL_IDENTITY"),
    )
    .where {
      (AccountsSubquery.mappingType eq 0) or
          ((AccountsSubquery.mappingType eq 2) and (AccountsSubquery.tag eq clients.accountTag)) or
          ((AccountsSubquery.mappingType eq 1) and (AccountsSubquery.alias eq clients.alias))
    }
```

Why can't we just do....

```
when {
  c.code == "EV" ->
    a.number.toString()
  else ->
    a.number.toString() + c.alias.substring(0, 2)
}
```

Why can't we just do....

```
fun condition(c: Client, a: Account): String =
  when {
    c.code == "EV" ->
      a.number.toString()
    else ->
      a.number.toString() + c.alias.substring(0, 2)
  }
```

Why can't we just do....

```
fun condition(c: Client, a: Account): String =
  when {
    c.code == "EV" ->
      a.number.toString()
    else ->
      a.number.toString() + c.alias.substring(0, 2)
  }
```

```
(c:Client, a:Account) -> String
```

That's right, we can't capture arbitrary expressions in runtime code...

whatsHappeningInside(

)

(c:Client, a:Account) -> String

whatsInside(Black Box)

Stringly DSL - Introspectable

```
americans
 .select($"firstName" as "called", $"lastName" as "alsoCalled", $"address_id" as "whereHeLives_id")
 .as("t")
 .join(addresses.as("a"), $"whereHeLivesId" === $"id")
 .select(
   concat(lit("Hello "), $"t.called", lit(" "), $"t.alsoCalled", lit(" of "), $"a.city"))
 .filter($"a.current" === lit(true))
```

.explain()

```
*(☝) Gimme My Result!  [concat(Hello , called,  , alsoCalled,  of , city)]
+- *(🤦) We're Joining! Huzzah! [whereHeLives_id], [id], Inner
  +- Join Key for the Left Side! (whereHeLives_id)
    +- *(1) Rename these like I said! Pronto! [firstName as Called... ]
      +- *(😇) I'm a smart format, load only: [firstName,lastName,address_id]
  +- Join Key for the Right Side! (id)
      +- *(😇) I'm a smart format, load only: [id,city,current]
               Read only current addr. from the file! 😎: [EqualTo(current,true)],
```

Closure DSL - NOT Introspectable

```
americans.map(a => HumanoidLivingSomewhere(a.firstName, a.lastName, a.addressId))
  .joinWith(addresses, $"id" === $"whereHeLivesId")
  .filter(ta => ta._2.current == true)
  .map { case (t, a) => s"Hello ${t.called} ${t.alsoCalled} of ${a.city}" }
```

.explain()

```
*(😱) Serialize Back Into a String Expensive!
+- *(3) Do the Outer Map that we Invoked
   +- (😱) Deserialize Tuple2 Expensive!
      +- 🤦 We're Joining! Huzzah! [_1.whereHeLives_id], [_2.id], Inner
        +- Join Key for the Left Side (_1.whereHeLives_id)
           +- *(1) Project [called, alsoCalled, whereHeLives_id]
              +- *(😱) Serialize the Join Key. Expensive!
                 +- *(1) MapElements HumanoidLivingSomewhere
                    +- (😱) Deserialize into a JVM Object (i.e. class American)
                       +- Scan All 'American' Columns Including 100 irrelevant ones!😱
        +- Join Key for the Right Side (_2.id)
           +- Scan All 'Address' Columns Including 100 irrelevant ones! 😱
              We Need to Read The Entire Dataset! No Excluding Non-Current Addresses 😢
```

whatsInside(

)

(c:Client, a:Account) -> String

Cannot use arbitrary T as a DSL because:

1. It is impossible to know what is inside.

whatsInside(

)

(c:Client, a:Account) -> String

Cannot use arbitrary T as a DSL because:

1. It is impossible to know what is inside.
2. ...

whatsInside(

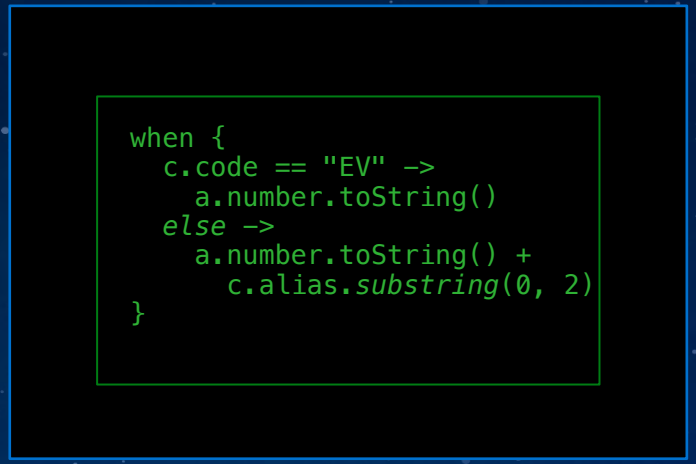
```
when(c) {
  is AbitraryType -> doSomethingArbitrary(c)
  else ->
    a.number.toString() + c.alias.substring(0, 2)
}
```

)

(c:Client, a:Account) -> String

Cannot use arbitrary T as a DSL because:

1. It is impossible to know what is inside.
2. Even if it were possible, how do you know it would convert successfully?

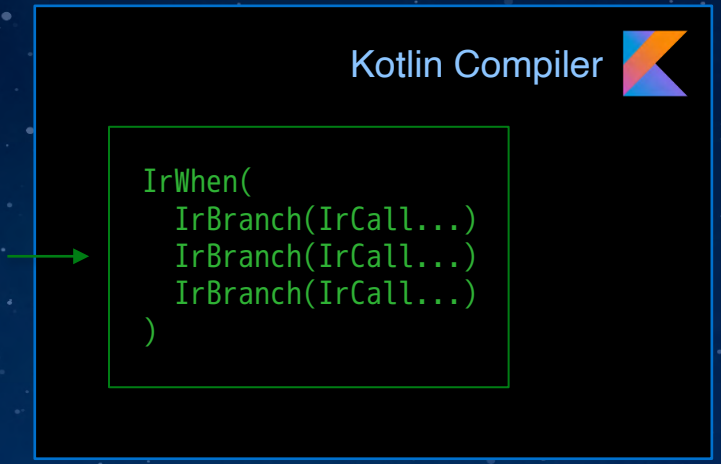
whatsInside()

```
when {
  c.code == "EV" ->
    a.number.toString()
  else ->
    a.number.toString() +
      c.alias.substring(0, 2)
}
```

(c:Client, a:Account) -> String

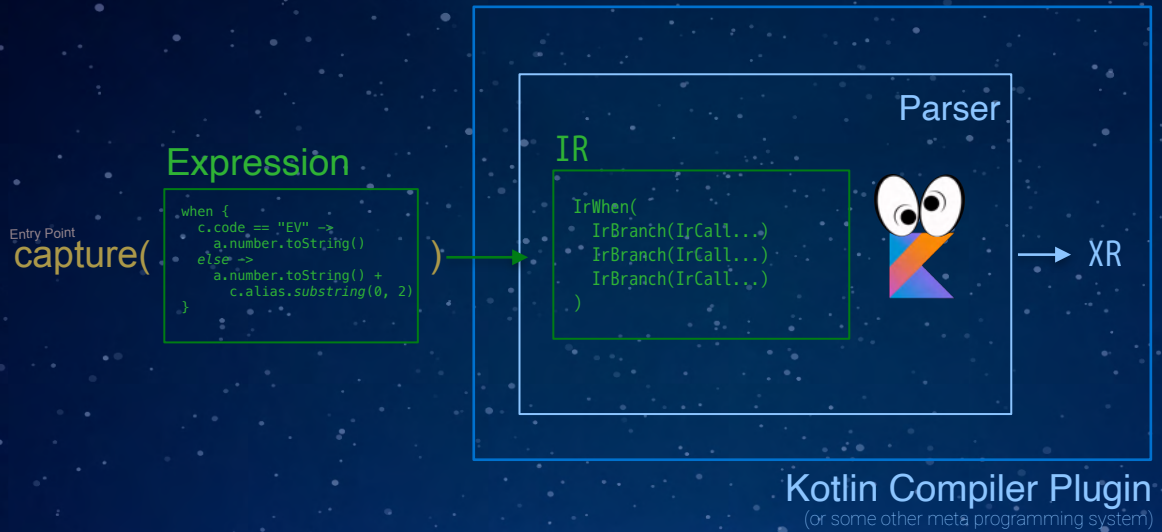
Expression

```
when {
  c.code == "EV" ->
    a.number.toString()
  else ->
    a.number.toString() +
      c.alias.substring(0, 2)
}
```



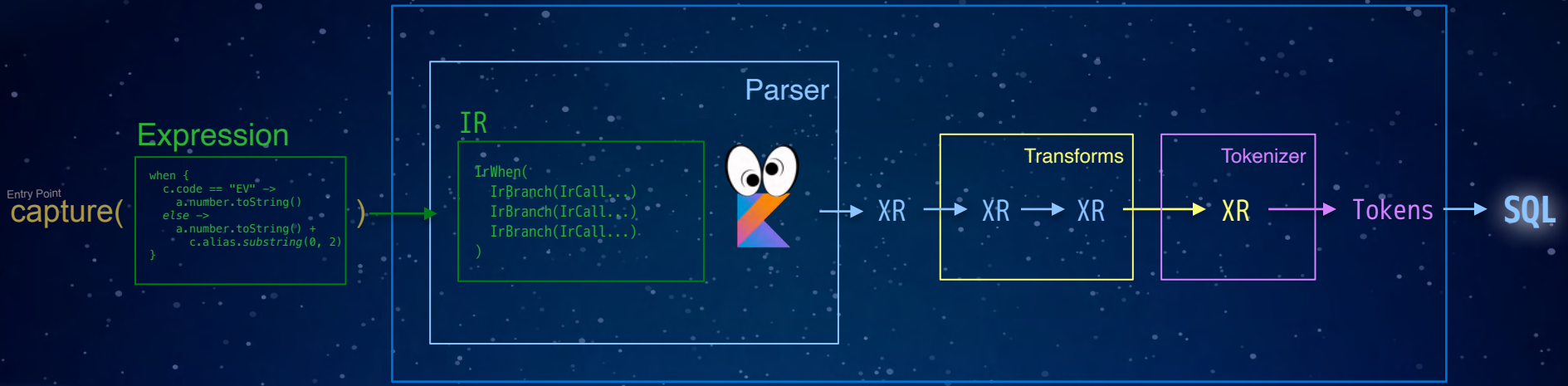
(c:Client, a:Account) -> String

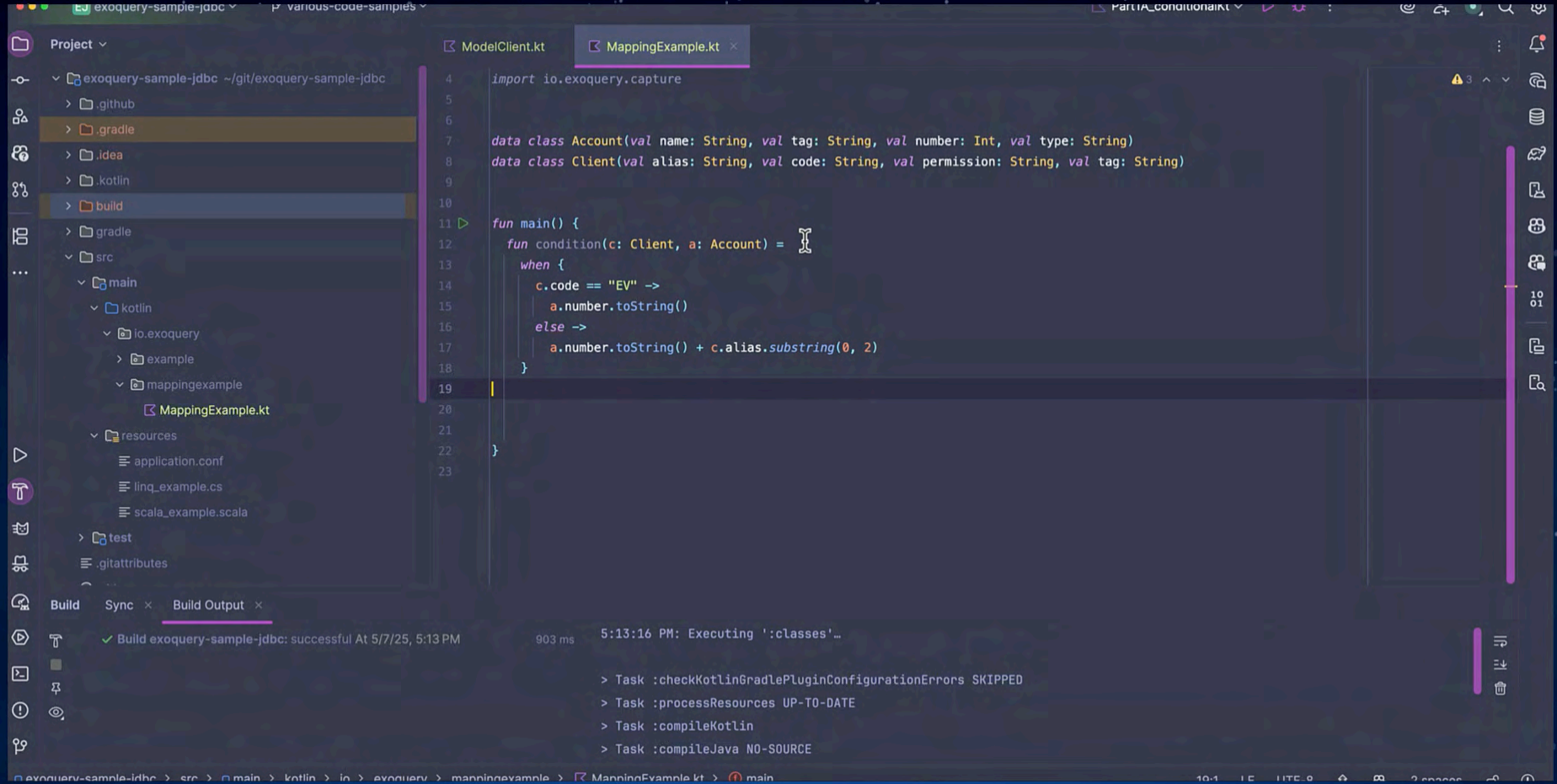
Quoted-DSL := A DSL Captured at Compile Time

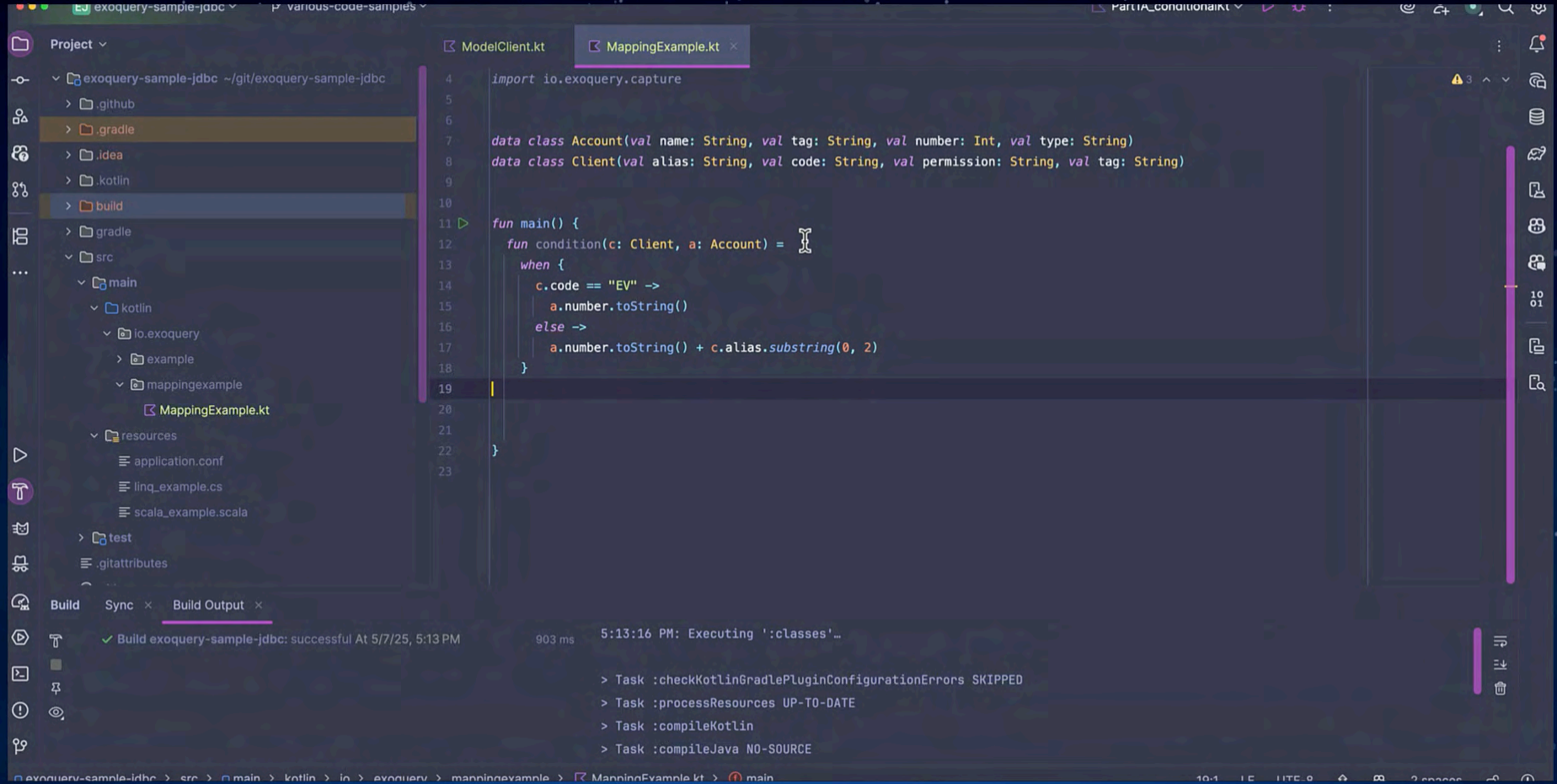


Quoted-DSL := A DSL Captured at Compile Time

...and transformed into something useful!




exoquery-sample-jdbc
various-code-samples
Project
ModelClient.kt
MappingExample.kt
exoquery-sample-jdbc ~/git/exoquery-sample-jdbc
.github
.gradle
.idea
.kotlin
build
gradle
src
main
kotlin
io.exoquery
example
mappingexample
MappingExample.kt
resources
application.conf
linq_example.cs
scala_example.scala
test
.gitattributes
import io.exoquery.capture
data class Account(val name: String, val tag: String, val number: Int, val type: String)
data class Client(val alias: String, val code: String, val permission: String, val tag: String)
fun main() {
fun condition(c: Client, a: Account) =
when {
c.code == "EV" ->
a.number.toString()
else ->
a.number.toString() + c.alias.substring(0, 2)
}
}
Build
Sync
Build Output
Build exoquery-sample-jdbc: successful At 5/7/25, 5:13 PM
903 ms
5:13:16 PM: Executing ':classes'…
> Task :checkKotlinGradlePluginConfigurationErrors SKIPPED
> Task :processResources UP-TO-DATE
> Task :compileKotlin
> Task :compileJava NO-SOURCE


exoquery-sample-jdbc
various-code-samples
Project
ModelClient.kt
MappingExample.kt
exoquery-sample-jdbc ~/git/exoquery-sample-jdbc
.github
.gradle
.idea
.kotlin
build
gradle
src
main
kotlin
io.exoquery
example
mappingexample
MappingExample.kt
resources
application.conf
linq_example.cs
scala_example.scala
test
.gitattributes
import io.exoquery.capture
data class Account(val name: String, val tag: String, val number: Int, val type: String)
data class Client(val alias: String, val code: String, val permission: String, val tag: String)
fun main() {
fun condition(c: Client, a: Account) =
when {
c.code == "EV" ->
a.number.toString()
else ->
a.number.toString() + c.alias.substring(0, 2)
}
}
Build
Sync
Build Output
Build exoquery-sample-jdbc: successful At 5/7/25, 5:13 PM
903 ms
5:13:16 PM: Executing ':classes'…
> Task :checkKotlinGradlePluginConfigurationErrors SKIPPED
> Task :processResources UP-TO-DATE
> Task :compileKotlin
> Task :compileJava NO-SOURCE

```
@CapturedFunction
fun condition(c: Client, a: Account) = capture.expression {
  when {
    c.code == "EV" ->
      a.number.toString()
    else ->
      a.number.toString() + c.alias.substring(0, 2)
  }
}

val query =
  capture.select {
    val c = from(Table<Client>())
    val a = join(Table<Account>()) { a -> a.tag == c.tag }
    Pair(
      c.alias,
      condition(c, a)
    )
  }
```

```
@CapturedFunction
fun condition(c: Client, a: Account) = capture.expression {
  when {
    c.code == "EV" ->
      a.number.toString()
    else ->
      a.number.toString() + c.alias.substring(0, 2)
  }
}

val query =
  capture.select {
    val c = from(Table<Client>())
    val a = join(Table<Account>()) { a -> a.tag == c.tag }
    Pair(
      c.alias,
      condition(c, a)
    )
  }
```

```
val query =
  capture.select {
    val c = from(Table<Client>())
    val a = join(Table<Account>()) { a -> a.tag == c.tag }
    Pair(
      c.alias,
      when {
        c.code == "EV" ->
          a.number.toString()
        else ->
          a.number.toString() + c.alias.substring(0, 2)
      }
    )
  }
```

How did we get here?

```
data class Account(val name: String, val tag: String, val number: Int, val type: String)
data class Client(val alias: String, val code: String, val permission: String, val tag: String)
```

```
val query =
  capture.select {
    val c = from(Table<Client>())
    val a = join(Table<Account>()) { a -> a.tag == c.tag }
    Pair(
      c.alias,
      when {
        c.code == "EV" ->
          a.number.toString()
        else ->
          a.number.toString() + c.alias.substring(0, 2)
      }
    )
  }
```

```
SELECT
  c.alias AS first,
  CASE
    WHEN c.code = 'EV' THEN a.number
    ELSE a.number || SUBSTRING(c.alias, 0, 2)
  END AS second
FROM
  Client c
  INNER JOIN Account a ON a.tag = c.tag
```

```
var query =
    from p in people
    join a in addresses on p.id == a.ownerId
    select new { p.name, a.street };
```

NO LEXER

```
var query =
    from p in people
    join a in addresses on p.id == a.ownerId
    select new { p.name, a.street };
```

LINQ

Microsoft .NET

```
val query = quote {
    for {
      p <- people
      a <- addresses.join(a -> p.id == a.ownerId)
    } yield (p.name, a.street)
}
```

NO FOR COMPREHENSION

```
val query = quote {
  for {
    p <- people
    a <- addresses.join(a -> p.id == a.ownerId)
  } yield (p.name, a.street)
}
```

```
val query = capture {
  select {
    val p = from(people)
    val a = join(addresses) { a -> p.id == a.ownerId) }
    p.name to a.street
  }
}
```

Shortened Form

```
val query =
  capture.select {
    val p = from(people)
    val a = join(addresses) { a -> p.id == a.ownerId) }
    p.name to a.street
  }
```

Some sort of gimmick?

```
val query =
  capture.select {
    val p = from(people)
    val a = join(addresses) { a -> p.id == a.ownerId) }
    p.name to a.street
  }
```

NO. IT'S REAL

```
var query = from p in people
            join a in addresses on p.id == a.ownerId
            select new { p.name, a.street };
```

```
var query = from p in people:Enumerable<Person>
            join a in addresses:Enumerable<Address> on p.id == a.ownerId
            select new { p.name, a.street };
```

```
class Person { int id; string name }
class Address { int ownerId; string street }
```

```
var query = from p:Person in people:Enumerable<Person>
            join a:Address in addresses:Enumerable<Person> on p.id == a.ownerId
            select new { p.name, a.street };
```

```
class Person { int id; string name }
class Address { int ownerId; string street }
```

```
var query = from p in people
            join a in addresses on p.id:int == a.ownerId:int
            select new { p.name:string, a.street:string };
```

```
class Person { int id; string name }
class Address { int ownerId; string street }
```

EXPANSION

```
var query = from p in people
            join a in addresses on p.id == a.ownerId
            select new { p.name, a.street };
```

SUGAR

```
var query =
  people.SelectMany(p =>
    addresses.Where(a => p.id == a.ownerId).Select(a =>
      new { x.p.name, x.a.street }
  )
```

EXPANSION

```
var query = from p in people
            join a in addresses on p.id == a.ownerId
            join r in robots on p.id == r.ownerId
            select new { p.Name, a.Street };
```

LINQ

Microsoft .NET

SUGAR

```
val query =
  people.SelectMany(p =>
    addresses.Where(a => p.id == a.ownerId).SelectMany(a =>
      robots.Where(r => p.id == r.ownerId).Select(r =>
        new { x.p.name, x.a.street, x.r.model }
      )
    )
  )
```

EXPANSION

```
var query = from p in people
            join a in addresses on p.id == a.ownerId
            join r in robots on p.id == r.ownerId
            join c in cars on p.id == c.ownerId
            select new { p.Name, a.Street };
```

LINQ

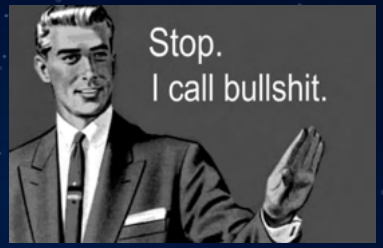
Microsoft .NET

SUGAR

```
val query =
  people.SelectMany(p =>
    addresses.Where(a => p.id == a.ownerId).SelectMany(a =>
      robots.Where(r => p.id == r.ownerId).SelectMany(r =>
        cars.Where(c => p.id == c.ownerId).Select(c =>
          new { p.name, a.street, r.model, c.make }
        )
    )
  )
)
```

EXPANSION

```
var query = from p in people
            join a in addresses on p.id == a.ownerId
            join r in robots on p.id == r.ownerId
            join c in cars on p.id == c.ownerId
            select new { p.Name, a.Street };
```



SUGAR

```
val query =
  people
    .SelectMany(p => addresses).Where((p, a) => p.id == a.ownerId)
    .SelectMany((p, a) => robots).Where((p, a, r) => p.id == r.ownerId)
    .SelectMany((p, a, r) => cars).Where((p, a, r, c) => p.id == c.ownerId)
    .Select((p, a, r, c) => new { p.name, a.street, r.model, c.make })
```

They’re not Nested in LINQ!

EXPANSION

```
val query =
  people.SelectMany(p =>
    addresses.Where(a => p.id == a.ownerId).SelectMany(a =>
      robots.Where(r => p.id == r.ownerId).SelectMany(r =>
        cars.Where(c => p.id == c.ownerId).Select(c =>
          new { x.p.name, x.a.street, x.r.model, x.c.make }
        )
    )
  )
)
```

SUGAR

```
val query =
  people
    .SelectMany(p => addresses).Where((p, a) => p.id == a.ownerId)
    .SelectMany((p, a) => robots).Where((p, a, r) => p.id == r.ownerId)
    .SelectMany((p, a, r) => cars).Where((p, a, r, c) => p.id == c.ownerId)
    .Select((p, a, r, c) => new { p.name, a.street, r.model, c.make })
```

EXPANSION

```
val query:Enumerable<Tuple> =
  people:Enumerable<Person>.SelectMany(p =>
    addresses.Where(a => p.id == a.ownerId):Enumerable<Address>.SelectMany(a =>
      robots.Where(r => p.id == r.ownerId):Enumerable<Robot>.SelectMany(r =>
        cars.Where(c => p.id == c.ownerId):Enumerable<Car>.Select(c =>
          new Tuple { x.p.name, x.a.street, x.r.model, x.c.make }
        )
    )
  )
)
```

SUGAR

```
val query:Enumerable<Tuple> =
  people:Enumerable<Person>
    .SelectMany(p => addresses).Where((p, a) => p.id == a.ownerId):Enumerable<Address>
    .SelectMany((p, a) => robots).Where((p, a, r) => p.id == r.ownerId):Enumerable<Robot>
    .SelectMany((p, a, r) => cars).Where((p, a, r, c) => p.id == c.ownerId):Enumerable<Car>
    .Select((p, a, r, c) => new Tuple { p.name, a.street, r.model, c.make })
```

```
val query =
  people.SelectMany(p =>
    addresses.Where(a => p.id == a.ownerId).SelectMany(a =>
      robots.Where(r => p.id == r.ownerId).SelectMany(r =>
        cars.Where(c => p.id == c.ownerId).Select(c =>
          new { p.name, a.street, r.model, c.make }
        )
    )
  )
)
```

```
val query =
  people.flatMap(p =>
    addresses.filter(a => p.id == a.ownerId).flatMap(a =>
      robots.filter(r => p.id == r.ownerId).flatMap(r =>
        cars.filter(c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
    )
  )
)
```

EXPANSION

```
val query =
  people.flatMap(p =>
    addresses.filter(a => p.id == a.ownerId).flatMap(a =>
      robots.filter(r => p.id == r.ownerId).flatMap(r =>
        cars.filter(c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
    )
  )
)
```

SUGAR

```
var query =
  for {
    p <- people
    a <- addresses if (p.id == a.ownerId)
    r <- robots if (p.id == r.ownerId)
    c <- cars if (p.id == c.ownerId)
  } yield (p.name, a.street, r.model, c.model)
```

EXPANSION

```
val query =
  people.flatMap(p =>
    addresses.join(a => p.id == a.ownerId).flatMap(a =>
      robots.join(r => p.id == r.ownerId).flatMap(r =>
        cars.join(c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
    )
  )
)
```

SUGAR

```
var query =
  for {
    p <- people
    a <- addresses.join(a => p.id == a.ownerId)
    r <- robots.join(r => p.id == r.ownerId)
    c <- cars.join(c => p.id == c.ownerId)
  } yield (p.name, a.street, r.model, c.model)
```

EXPANSION

```
val query =
  people.flatMap(p =>
    addresses.join(a => p.id == a.ownerId).flatMap(a =>
      robots.join(r => p.id == r.ownerId).flatMap(r =>
        cars.joinLeft(c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
    )
  )
)
```

SUGAR

```
var query =
  for {
    p <- people                                    //p: Person
    a <- addresses.join(a => p.id == a.ownerId)    //a: Address
    r <- robots.join(r => p.id == r.ownerId)       //r: Robot
    c <- cars.joinLeft(c => p.id == c.ownerId)     //c: Option[Car]
  } yield (p.name, a.street, r.model, c.model)
```

No for-comprehensions in 


```
var query =
  ??? {
    val p = people.bind()
    val a = addresses.join(a => p.id == a.ownerId).bind()
    val r = robots.join(r => p.id == r.ownerId).bind()
    val c = cars.joinLeft(c => p.id == c.ownerId).bind()
    (p.name, a.street, r.model, c.model)
  }
```

Monadless

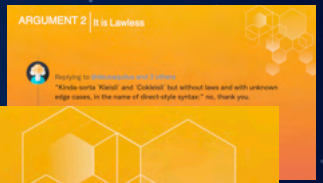
Isomorphic!

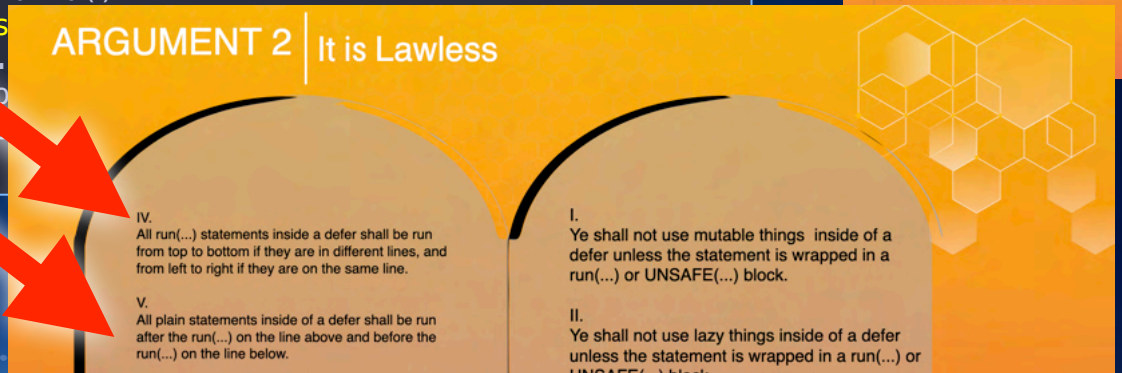
```
var query =
  ??? {
    val p = people.bind()
    val a = addresses.join(a => p.id == a.ownerId).bind()
    val r = robots.join(r => p.id == r.ownerId).bind()
    val c = cars.joinLeft(c => p.id == c.ownerId).bind()
    (p.name, a.street, r.model, c.model)
  }
```

Monadless

Isomorphic!

```
var query =
  ??? {
    val p = people.bind()
    val a = address
    val r =  ts.
    val c = cars
    (p.name, a.str
  }
```





EXPANSION

```
val query =
  people.flatMap(p =>
    addresses.join(a => p.id == a.ownerId).flatMap(a =>
      robots.join(r => p.id == r.ownerId).flatMap(r =>
        cars.joinLeft(c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
    )
  )
)
```

Isomorphic!

SUGAR

```
var query =
  ??? {
    val p = people.bind()
    val a = addresses.join(a => p.id == a.ownerId).bind()
    val r = robots.join(r => p.id == r.ownerId).bind()
    val c = cars.joinLeft(c => p.id == c.ownerId).bind()
    (p.name, a.street, r.model, c.model)
  }
```

EXPANSION

```
val query =
  people.flatMap(p =>
    addresses.join(a => p.id == a.ownerId).flatMap(a =>
      robots.join(r => p.id == r.ownerId).flatMap(r =>
        cars.joinLeft(c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
    )
  )
)
```

Direct Style `.bind()` -> cut-point

SUGAR

```
var query =
  ??? {
    val p = people:SqlQuery<Person>.bind():Person
    val a = addresses.join(a => p.id == a.ownerId):SqlQuery<Address>.bind():Address
    val r = robots.join(r => p.id == r.ownerId):SqlQuery<Robot>.bind():Robot
    val c = cars.joinLeft(c => p.id == c.ownerId):SqlQuery<Car>.bind():Car
    (p.name, a.street, r.model, c.model)
  }
```

EXPANSION

```
val query =
  people.flatMap(p =>
    addresses.join(a => p.id == a.ownerId).flatMap(a =>
      robots.join(r => p.id == r.ownerId).flatMap(r =>
        cars.joinLeft(c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
    )
  )
)
```

SUGAR

```
var query =
  ??? {
    val p = people:SqlQuery<Person>.bind():Person
    val a = addresses.join(a => p.id == a.ownerId):SqlQuery<Address>.bind()->:Address
    val r = robots.join(r => p.id == r.ownerId):SqlQuery<Robot>.bind()->:Robot
    val c = cars.joinLeft(c => p.id == c.ownerId):SqlQuery<Car>.bind()->:Car
    (p.name, a.street, r.model, c.model)
  }
```

EXPANSION

```
val query =
  people.flatMap(p =>
    addresses.join(a => p.id == a.ownerId).flatMap(a =>
      robots.join(r => p.id == r.ownerId).flatMap(r =>
        cars.joinLeft(c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
    )
  )
)
```

SUGAR

EXPANSION

```
val query =
  people.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
     )
  )
)
```

from DSL flatMap!

SUGAR

```
var query =
  capture.select {
    val p = from(people)
    val a = join(addresses) { a => p.id == a.ownerId }
    val r = join(robots)    { r => p.id == r.ownerId }
    val c = joinLeft(cars)  { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

EXPANSION

```
val query =
  people.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
  )
)
```

join DSL does both!

SUGAR

```
var query =
  capture.select {
    val p = from(people)
    val a = join(addresses) { a => p.id == a.ownerId }
    val r = join(robots)    { r => p.id == r.ownerId }
    val c = joinLeft(cars)  { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

EXPANSION

```
val query =
  people.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
  )
)
```

SUGAR

```
var query =
  capture.select {
    val p:Person  = from(people)
    val a:Address = join(addresses) { a => p.id == a.ownerId }
    val r:Robot   = join(robots)    { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars)  { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```


```
data class Person(val id: Int, val name: String, val age: Int)
data class Address(val ownerId: Int, val street: String, val zip: Int)
data class Robot(val ownerId: Int, val model: String)
data class Car(val ownerId: Int, val make: String, val model: String)
```

EXPANSION

```
val query =
  people.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
  )
)
```

SUGAR

```
var query =
  capture.select {
    val p:Person  = from(people)
    val a:Address = join(addresses) { a => p.id:Int == a.ownerId:Int }
    val r:Robot   = join(robots)    { r => p.id:Int == r.ownerId:Int }
    val c:Car     = joinLeft(cars)  { c => p.id:Int == c.ownerId:Int }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
data class Person(val id: Int, val name: String, val age: Int)
data class Address(val ownerId: Int, val street: String, val zip: Int)
data class Robot(val ownerId: Int, val model: String)
data class Car(val ownerId: Int, val make: String, val model: String)
```

EXPANSION

```
val query =
  people.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
  )
)
```

SUGAR

```
var query:SqlQuery<Tuple> =
  capture.select {
    val p:Person  = from(people:SqlQuery<Person>)
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
data class Person(val id: Int, val name: String, val age: Int)
data class Address(val ownerId: Int, val street: String, val zip: Int)
data class Robot(val ownerId: Int, val model: String)
data class Car(val ownerId: Int, val make: String, val model: String)
```

```
SELECT
  p.id, p.name, p.age, c.ownerId, c.make, c.model
FROM
  Person p
  INNER JOIN Address a ON a.ownerId = p.id
  INNER JOIN Robot r ON r.ownerId = p.id
  INNER JOIN Car c ON c.ownerId = p.id
```

The DE-SUGARED form becomes SQL...
so the system composes!

EXPANSION

SUGAR

```
val query =
  people.map { p -> … p }.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
  )
)
```

```
var query =
  capture.select {
    val p  = from(
      capture {
        people.map { p -> … p }
      }
    )
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val query =
  people.filter { p -> … p }.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
  )
)
```

```
var query =
  capture.select {
    val p  = from(
      capture {
        people.filter { p -> … p }
      }
    )
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val query =
  people.filter { p -> … p }.distinct().flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
  )
)
```

```
var query =
  capture.select {
    val p  = from(
      capture {
        people.filter { p -> … p }.distinct()
      }
    )
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val query =
  (people.filter { p -> … p } union people.filter { p -> … p }).flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
     )
  )
)
```

```
var query =
  capture.select {
    val p  = from(
      capture { people.filter { p -> … p }… }
      union
      capture { otherPeople.filter { p -> … p }… }
    )
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val query =
  (people.filter { p -> … p } union people.filter { p -> … p }).flatMap(p =>
   flatJoin(addresses.filter { a -> … a }.distinct(), a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots.map { r -> … r }, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars.distinct(), c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
        )
     )
  )
)
```

Compose any or all Constructs!

```
var query =
  capture.select {
    val p  = from(
      capture { people.filter { p -> … p }… }
      union
      capture { otherPeople.filter { p -> … p }… }
    )
    val a:Address = join(addresses.filter { a -> … a }.distinct()) { a => p.id == a.ownerId }
    val r:Robot   = join(robots.map { r -> … r }) { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars.distinct()) { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val query =
  people.flatMap {
   ...map(… => p)
  }.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
   )
)
```

```
SELECT
  p.id,
  p.name,
  p.age,
  c.ownerId AS second,
  c.make AS first,
  c.model AS second
FROM
  Person p
  ...
  INNER JOIN Address a ON a.ownerId = p.id
  INNER JOIN Robot r ON r.ownerId = p.id
  INNER JOIN Car c ON c.ownerId = p.id
```

Auto Flattening the 1st Clause!

```
var query =
  capture.select {
    val p  = from(
      capture.select {
        val p = from(people)
        …
        p
      }
    )
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val query =
  people.flatMap {
    flatJoin(vips) { v ->
      v.id == p.id
    }.map { v -> p }
  }.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
  )
)
```

```
SELECT
  p.id,
  p.name,
  p.age,
  c.ownerId AS second,
  c.make AS first,
  c.model AS second
FROM
  Person p
  INNER JOIN Vip v ON v.id = p.id
  INNER JOIN Address a ON a.ownerId = p.id
  INNER JOIN Robot r ON r.ownerId = p.id
  INNER JOIN Car c ON c.ownerId = p.id
```

Auto Flattening the 1st Clause!

```
var query =
  capture.select {
    val p  = from(
      capture.select {
        val p = from(people)
        val v = join(vips) { v -> v.id == p.id }
        p
      }
    )
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val query =
  people.flatMap {
    flatJoin(vips) { v ->
      v.id == p.id
    }.map { v -> p }
  }.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
   )
)
```

```
SELECT
  p.id,
  p.name,
  p.age,
  c.ownerId AS second,
  c.make AS first,
  c.model AS second
FROM
  Person p
  INNER JOIN Vip v ON v.id = p.id
  INNER JOIN Address a ON a.ownerId = p.id
  INNER JOIN Robot r ON r.ownerId = p.id
  INNER JOIN Car c ON c.ownerId = p.id
```

Auto Flattening the 1st Clause!

```
var query =
  capture.select {
    val p  = from(
      capture.select {
        ...
        capture.select {
          val p = from(people)
          val v = join(vips) { v -> v.id == p.id }
          p
        }
      }
    )
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val query =
  people.flatMap {
    A.flatMap {
      B.flatMap {
        C.flatMap {
          ...
        }
      }
    }
  }.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
    flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
     flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
        ( p.name, a.street, r.model, c.make )
      )
    )
  )
)
```

```
SELECT
  p.id,
  p.name,
  p.age,
  c.ownerId AS second,
  c.make AS first,
  c.model AS second
FROM
  Person p
  INNER JOIN ...
  INNER JOIN ...
  INNER JOIN ...
  INNER JOIN ...
  INNER JOIN Address a ON a.ownerId = p.id
  INNER JOIN Robot r ON r.ownerId = p.id
  INNER JOIN Car c ON c.ownerId = p.id
```

Auto Flattening the 1st Clause!

```
var query =
  capture.select {
    val p  = from(
      capture.select {
        capture.select {
          capture.select {
            capture.select {
              capture.select {
                ...
              }
            }
          }
        }
      }
    )
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val a = A.flatMap { ... }
val b = B.flatMap { ... a ... }
val c = C.flatMap { ... b ... }

val query =
  people.flatMap {
    ... c ...
  }.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
   )
)
```

```
SELECT
  p.id,
  p.name,
  p.age,
  c.ownerId AS second,
  c.make AS first,
  c.model AS second
FROM
  Person p
  INNER JOIN ...
  INNER JOIN ... a ...
  INNER JOIN ... b ...
  INNER JOIN ... c ...
  INNER JOIN Address a ON a.ownerId = p.id
  INNER JOIN Robot r ON r.ownerId = p.id
  INNER JOIN Car c ON c.ownerId = p.id
```

Auto Flattening the 1st Clause!

```
val a = capture.select {
  ...
}
val b = capture.select {
  ... a ...
}
val c = capture.select {
  ... b ...
}

var query =
  capture.select {
    val p  = from(c)
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val a = A.flatMap { ... }
val b = B.flatMap { ... a ... }
val c = C.flatMap { ... b ... }

val query =
  people.flatMap {
    ... c ...
  }.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
     flatJoin(robots, r => p.id == r.ownerId).flatMap(r =>
       flatJoinLeft(cars, c => p.id == c.ownerId).map(c =>
          ( p.name, a.street, r.model, c.make )
       )
     )
   )
)
```

```
SELECT
  p.id,
  p.name,
  p.age,
  c.ownerId AS second,
  c.make AS first,
  c.model AS second
FROM
  Person p
  INNER JOIN ...
  INNER JOIN ... a ...
  INNER JOIN ... b ...
  INNER JOIN ... c ...
  INNER JOIN Address a ON a.ownerId = p.id
  INNER JOIN Robot r ON r.ownerId = p.id
  INNER JOIN Car c ON c.ownerId = p.id
```

Auto Flattening the 1st Clause!

```
val a = capture.select {
  ...
}
val b = capture.select {
  ... a ...
}
val c = capture.select {
  ... b ...
}
```

Logically Equivalent

```
capture.select {
  capture.select {
    capture.select {
      capture.select {
        ...
      }
    }
  }
}
```

```
var query =
  capture.select {
    val p  = from(c)
    val a:Address = join(addresses:SqlQuery<Address>) { a => p.id == a.ownerId }
    val r:Robot   = join(robots:SqlQuery<Robot>)      { r => p.id == r.ownerId }
    val c:Car     = joinLeft(cars:SqlQuery<Car>)      { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

The Flatter, the Better

Query Compilation Based on the Flattening Transformation

Alexander Ulrich Torsten Grust
Universität Tübingen
Tübingen, Germany
[alexander.ulrich, torsten.grust]@uni-tuebingen.de

ABSTRACT

We demonstrate the insides and outs of a query compiler based on the **flattening transformation**, a translation technique designed by the programming language community to derive efficient data-parallel implementations from iterative programs. Flattening admits the straightforward formulation of intricate query logic—including deeply nested loops over (possibly ordered) data or the construction of rich data structures. To demonstrate the level of expressiveness that

In tandem with established query compilation techniques that we repeatedly draw on here, flattening enables a principled translation that proceeds in small digestible steps—a welcome advance over rather complex and monolithic approaches, including our own [7]. Flattening admits rich user-facing query languages and data models beyond those offered by recent related efforts (*query shredding* [4], for example, lacks support for ordered data, grouping, or aggregation). To make these points we express the examples in

```
SELECT
  p.id,
  p.name,
  p.age,
  c.ownerId AS second,
  c.make AS first,
  c.model AS second
FROM
  Person p
  INNER JOIN ...
  INNER JOIN ... a ...
  INNER JOIN ... b ...
  INNER JOIN ... c ...
  INNER JOIN Address a ON a.ownerId = p.id
  INNER JOIN Robot r ON r.ownerId = p.id
  INNER JOIN Car c ON c.ownerId = p.id
```

Nested Constructs -> Flat Query -> Better Performance!

```
val a = capture.select {
  ...
}
val b = capture.select {
  ... a ...
}
val c = capture.select {
  ... b ...
}

var query =
  capture.select {
    val p  = from(c)
    val a:Address = join(addresses:SqlQuery<Address>)
    val r:Robot   = join(robots:SqlQuery<Robot>)
    val c:Car     = joinLeft(cars:SqlQuery<Car>)
    Tuple(p.name, a.street, r.model, c.model)
  }
```

```
val query =
  people.flatMap(p =>
   flatJoin(addresses, a => p.id == a.ownerId).flatMap(a =>
      ( p.name, ..., a.street, ... )
    )
  )
)
```

```
SELECT
  p.id,
  p.name,
  p.age,
  a.ownerId,
  a.street,
  a.city,
  a.zip
FROM Person p
INNER JOIN Address a
ON a.ownerId = this.id
```

Domain-Specific Join

```
@CapturedFunction
fun Person.joinAddresses() = capture {
  internal.flatJoin(Table<Address>()) { a -> a.ownerId ==  this@joinAddresses.id }
}

val query = capture.select {
  val p = from(Table<Person>())
  val a = from(p.joinAddresses())
  Pair(p, a)
}
```

ExoQuery
ADVANCED
TOPIC

info@exoquery.io

```
SELECT
  p.id,
  p.name,
  p.age,
  a.ownerId,
  a.street,
  a.city,
  a.zip
FROM Person p, Address a
```

Domain-Specific Join

```
@CapturedFunction
fun Person.joinAddresses() = capture {
  internal.flatJoin(Table<Address>()) { a -> a.ownerId ==  this@joinAddresses.id }
}

val query = capture.select {
  val p = from(Table<Person>())
  val a = from(Table<Address>())
  Pair(p, a)
}
```

ExoQuery

ADVANCED TOPIC

info@exoquery.io

The Power of LINQ is Composition

ExoQuery is That

```
var query =
  capture.select {
    val p = from(people)
    val a = join(addresses) { a => p.id == a.ownerId }
    val r = join(robots)    { r => p.id == r.ownerId }
    val c = joinLeft(cars)  { c => p.id == c.ownerId }
    Tuple(p.name, a.street, r.model, c.model)
  }
```

What are all those features you mentioned?

```
val query =
  capture.select {
    val p = from(people)
    val a = join(addresses) { a -> p.id == a.ownerId) }
    p.name to a.street
  }
```

What are **people** and **addresses**?

```
val people = capture { Table<Person>() }
val addresses = capture { Table<Address>() }
```

```
val query =
  capture.select {
    val p = from(people)
    val a = join(addresses) { a -> p.id == a.ownerId) }
    p.name to a.street
  }
```

What are **people** and **addresses**?

```
val people = capture { Table<Person>():SqlQuery<Person> }
val addresses = capture { Table<Address>():SqlQuery<Person> }
```

```
val query =
  capture.select {
    val p = from(people)
    val a = join(addresses) { a -> p.id == a.ownerId) }
    p.name to a.street
  }
```

What are **people** and **addresses**?

```
val people:SqlQuery<Person> = capture { Table<Person>() }
val addresses:SqlQuery<Person> = capture { Table<Address>() }
```

```
val query =
  capture.select {
    val p = from(people)
    val a = join(addresses) { a -> p.id == a.ownerId) }
    p.name to a.street
  }
```

What are **people** and **addresses**?

```
val people = capture { Table<Person>() }
val addresses = capture { Table<Address>() }
```

```
fun <T> capture(block: CapturedBlock.() -> SqlQuery<T>): SqlQuery<T> = ...

interface CapturedBlock {
  fun <T> Table(): SqlQuery<T> = ...
}
```

You cannot declare Table<T> outside of a capture!

```
val query =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId) }
    p.name to a.street
  }

q.buildFor.Postgres()
```

```
SELECT p.name AS first, a.street AS second
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
```

```
val query:SqlQuery<Pair<String, String>> =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId) }
    p.name to a.street
  }

q.buildFor.Postgres()
```

```
SELECT p.name AS first, a.street AS second
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
```

```
val query:SqlQuery<Pair<String, String>> =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId) }
    p.name to a.street
  }

q.buildFor.Postgres().run(db):List<Pair<String, String>>
```

```
SELECT p.name AS first, a.street AS second
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
```

```
val query:SqlQuery<Pair<Person, Address>> =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId) }
    p to a
  }

q.buildFor.Postgres().run(db):List<Pair<Person, Address>>
```

```
SELECT p.id, p.name, p.age, a.ownerId, a.street, a.zip
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
```

```
val query =
  capture.select {
    val p = from(Table<Person>())
    val a = joinLeft(Table<Address>()) { a -> p.id == a.ownerId) }
    p to a
  }

q.buildFor.Postgres().run(db)
```

```
SELECT p.id, p.name, p.age, a.ownerId, a.street, a.zip
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
```

```
val query:SqlQuery<Pair<Person, Address?>> =
  capture.select {
    val p:Person   = from(Table<Person>())
    val a:Address? = joinLeft(Table<Address>()) { a -> p.id == a.ownerId) }
    p to a
  }

q.buildFor.Postgres().run(db):List<Pair<Person, Address?>>
```

```
SELECT p.id, p.name, p.age, a.ownerId, a.street, a.zip
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
```

```
val query:SqlQuery<Pair<Person, Address?>> =
  capture.select {
    val p:Person   = from(Table<Person>())
    val a:Address? = joinLeft(Table<Address>()) { a -> p.id == a.ownerId }
    p to a?.street
  }

q.buildFor.Postgres().run(db):List<Pair<Person, String?>>
```

```
SELECT p.id, p.name, p.age, a.street AS second
FROM Person p
LEFT JOIN Address a ON p.id = a.ownerId
```

```
val query:SqlQuery<Pair<Person, Address?>> =
  capture.select {
    val p:Person   = from(Table<Person>())
    val a:Address? = joinLeft(Table<Address>()) { a -> p.id == a.ownerId }
    p to a?.let { a -> a.street to a.zip }
  }

q.buildFor.Postgres().run(db):List<Pair<Person, Pair<String, Int>?>>
```

```
SELECT p.id, p.name, p.age, a.ownerId, a.street, a.zip
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
```

```
val query:SqlQuery<Pair<Person, Address?>> =
  capture.select {
    val p:Person   = from(Table<Person>())
    val a:Address? = joinLeft(Table<Address>()) { a -> p.id == a.ownerId }
    p to (a?.street ?: "default")
  }

q.buildFor.Postgres().run(db):List<Pair<Person, String>>
```

```
SELECT
  p.id, p.name, p.age,
  CASE WHEN a.street IS NULL THEN 'default'
    ELSE a.street
  END AS second
FROM Person p LEFT JOIN Address a ON p.id = a.ownerId
```

Supports Where

```
val query =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId) }
    where { p.age > 30 }
    p to a
  }
```

```
SELECT p.id, p.name, p.age, a.ownerId, a.street, a.zip
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
WHERE p.age > 30
```

...and GroupBy

```
val query =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId) }
    where { p.age > 30 }
    groupBy(p.name)
    Triple(p.name, avg(p.age), count(a.zip))
  }
```

```
SELECT p.name AS first, avg(p.age) AS second, count(a.zip) AS third
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
WHERE p.age > 30
GROUP BY p.name
```

...with multiple fields

```
val query =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId) }
    where { p.age > 30 }
    groupBy(p.name, p.name)
    Triple(p.name, p.age, count(a.zip))
  }
```

```
SELECT p.name AS first, p.age AS second, count(a.zip) AS third
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
WHERE p.age > 30
GROUP BY p.name, p.age
```

Supports SortBy, with Specified Orders

```
val query =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId) }
    where { p.age > 30 }
    groupBy(p.name, p.name)
    sortBy(p.name to Ord.Asc, p.age to Ord.Desc)
    Triple(p.name, p.age, count(a.zip))
  }
```

```
SELECT p.name AS first, p.age AS second, count(a.zip) AS third
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
WHERE p.age > 30
GROUP BY p.name, p.age
ORDER BY p.name ASC, p.age DESC
```

```
val query:SqlQuery<Triple<String, Int, Int>> =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId) }
    where { p.age > 30 }
    groupBy(p.name, p.name)
    sortBy(p.name to Ord.Asc, p.age to Ord.Desc)
    Triple(p.name, p.age, count(a.zip))
  }

q.buildFor.Postgres().run(db):List<Triple<String, Int, Int>>
```

```
SELECT p.name AS first, p.age AS second, count(a.zip) AS third
FROM Person p
INNER JOIN Address a ON p.id = a.ownerId
WHERE p.age > 30
GROUP BY p.name, p.age
ORDER BY p.name ASC, p.age DESC
```

```
val query:SqlQuery<Person> =
  capture.select {
    val p = from(Table<Person>())
    p
  }

q.buildFor.Postgres().run(db):List<Person>
```

```
SELECT x.id, x.name, x.age FROM Person x
```

Table<T> is just a **SqlQuery<T>** Constructor!

```
val query:SqlQuery<Person> =
  capture.select {
    val p = from(Table<Person>():SqlQuery<Person>)
    p
  }

q.buildFor.Postgres().run(db):List<Person>
```

```
SELECT x.id, x.name, x.age FROM Person x
```

```
val query:SqlQuery<Person> =
  capture {
   select {
      val p = from(Table<Person>())
      p
    }
  }

q.buildFor.Postgres().run(db):List<Person>
```

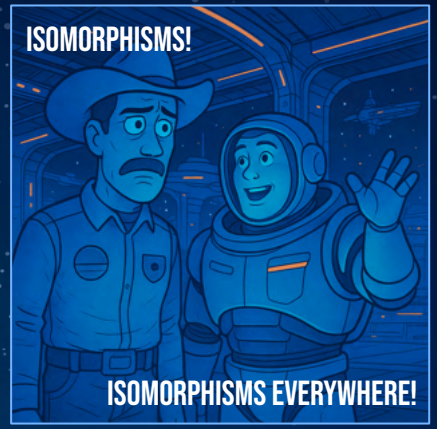
```
SELECT x.id, x.name, x.age FROM Person x
```

Isomorphisms Allow the system to be Composeable

```
val query:SqlQuery<Person> =
  capture {
   Table<Person>():SqlQuery<Person>
  }

q.buildFor.Postgres().run(db):List<Person>
```

```
SELECT x.id, x.name, x.age FROM Person x
```



Supports Filter (Applicative)

```
val query =
  capture {
    Table<Person>().filter { p -> p.name == "Joe" }
  }

query.buildFor.Postgres().value
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = 'Joe'
```

...which does Correlated Subqueries!

```
val query =
  capture {
    Table<Person>().filter { p ->
      p.age > Table<Person>().map { it.age }.avg()
    }
  }

query.buildFor.Postgres().value
```

```
SELECT p.id, p.name, p.age FROM Person p
WHERE p.age > (SELECT avg(it.age) FROM Person it)
```

Supports Map (a.k.a. Functor)

```
val query =
  capture {
    Table<Person>().map { p -> Pair(p.name, p.age) }
  }

query.buildFor.Postgres().value
```

```
SELECT p.name AS first, p.age AS second FROM Person p
```

...with Select-Position Aggregators

```
val query =
  capture {
    Table<Person>().map { p -> Pair(count(p.name), avg(p.age)) }
  }

query.buildFor.Postgres().value
```

```
SELECT count(p.name) AS first, avg(p.age) AS second FROM Person p
```

count: (String) -> String avg: (Int) -> Double

Supports Distinct

```
val query =
  capture {
    Table<Person>().map { p -> Pair(p.name, p.age) }.distinct()
  }

query.buildFor.Postgres().value
```

```
SELECT DISTINCT p.name AS first, p.age AS second FROM Person p
```

Supports Limit and Offset

```
val query =
  capture {
    Table<Person>().drop(1).take(1)
  }

query.buildFor.Postgres().value
```

```
SELECT x.id, x.name, x.age FROM Person x LIMIT 5 OFFSET 1
```

Supports Union

```
val query =
  capture {
    Table<Person>().filter { p -> p.name == "Joe" } union
      Table<Person>().filter { p -> p.name == "Jack" }
  }

query.buildFor.Postgres().value
```

```
(SELECT p.id, p.name, p.age FROM Person p WHERE p.name = 'Joe')
  UNION
(SELECT p1.id, p1.name, p1.age FROM Person p1 WHERE p1.name = 'Jack')
```

Useful Pattern: Map to Common Type, then Union

```
data class IdAndName(val id: Int, val name: String)

val query =
  capture {
    Table<Person>().map { p -> IdAndName(p.id, p.name) } union
      Table<Robot>().map { r -> IdAndName(r.id, r.model) }
  }


query.buildFor.Postgres().value
```

```
(SELECT p.id, p.name FROM Person p)
  UNION
(SELECT r.id, r.model AS name FROM Robot r)
```

Useful Pattern: Map to Common Type, then Union

```
val peopleInNewYork =
  capture.select {
    val p = from(Table<Person>())
    val a = join(Table<Address>()) { a -> p.id == a.ownerId }
    where { a.city == "New York" }
    p
  }

val tSeriesRobots =
  capture { Table<Robot>().filter { r -> r.model.substring(0, 2) == "T-" } }


data class IdAndName(val id: Int, val name: String)

val query =
  capture {
    peopleInNewYork.map { p -> IdAndName(p.id, p.name) } union
      tSeriesRobots.map { r -> IdAndName(r.id, r.model) }
  }


query.buildFor.Postgres().value
```

```
(SELECT p.id, p.name FROM Person p
 INNER JOIN Address a ON p.id = a.ownerId WHERE a.city = 'New York')
UNION
(SELECT r.id, r.model AS name FROM Robot r WHERE SUBSTRING(r.model, 0, 2) = 'T-')
```

Parameters

(a.k.a. wrestling with runtime data)

So all of this is with static values...

```
val query =
  capture {
    Table<Person>().filter { p -> p.name == "Joe" }
  }

query.buildFor.Postgres().value
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = 'Joe'
```

How do I get a runtime variable into the capture?

```
val runtimeName: String = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == <???> }
  }

query.buildFor.Postgres().value
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = <???>
```

Presumably you don't want SQL Injections...

```
val runtimeName: String = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == "Joe'; drop table Person;'" }
  }

query.buildFor.Postgres().value
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = 'Joe'; drop table Person;''
```

...so you need to propagate

```
val runtimeName: String = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == param(runtimeName) }
  }

query.buildFor.Postgres().value
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?
```

...all the way into the PreparedStatement

```
val runtimeName: String = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == param(runtimeName) }
  }

query.buildFor.Postgres().value
```

```
preparedStatement.set(1, runtimeName)
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?
```

How do I get a runtime variable into the capture?

```
val runtimeName: SomeArbitraryType = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == param(runtimeName) }
  }

query.buildFor.Postgres().value
```

```
preparedStatement.set___(1, runtimeName)
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?
```

How do I get a runtime variable into the capture?

```
val runtimeName: Email = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == param(runtimeName) }
  }

query.buildFor.Postgres().value
```

```
preparedStatement.set???(1, runtimeName)
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?
```

How do I get a runtime variable into the capture?

```
val runtimeName: Email = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == param(runtimeName) }
  }

query.buildFor.Postgres().value
```

kotlinx.serialization

```
preparedStatement.setParam(1, runtimeName)
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?
```

How do I get a runtime variable into the capture?

```
val runtimeName: Email = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == param(runtimeName) }
  }

query.buildFor.Postgres().value
```

Abstracts away PreparedStatement ...can use with non-JDBC!

```
object EmailSerializer : KSerializer<Email> {
  override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("Email", PrimitiveKind.STRING)
  override fun serialize(encoder: Encoder, value: Email) = encoder.encodeString(value.value)
  override fun deserialize(decoder: Decoder) = Email(decoder.decodeString())
}
```

```
preparedStatement.setParam(1, runtimeName)
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?
```

How do I get a runtime variable into the capture?

```
val runtimeName: Email = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == param(runtimeName) }
  }

query.buildFor.Postgres().value
```

```
object EmailSerializer : KSerializer<Email> {
  override val descriptor: SerialDescriptor = PrimitiveSerialDescriptor("Email", PrimitiveKind.STRING)
  override fun serialize(encoder: Encoder, value: Email) = encoder.encodeString(value.value)
  override fun deserialize(decoder: Decoder) = Email(decoder.decodeString())
}
```

```
preparedStatement.setParam(1, runtimeName)
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?
```

How do I get a runtime variable into the capture?

```
val runtimeName: Email = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == param(runtimeName) }
  }

query.buildFor.Postgres().value
```


```
val emailSerializer: KSerializer<Email> =
  serializer<String>().extend.to<Email>()
    .withMap { Email(it) }
    .withContramap { it.value }
    .build("Email")
```

```
preparedStatement.setParam(1, runtimeName)
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?
```

How do I get a runtime variable into the capture?

```
val runtimeName: Email = somethingFromSomewhere()

val query =
  capture {
    Table<Person>().filter { p -> p.name == paramCtx(runtimeName) }
  }

query.buildFor.Postgres().value
```

Low Level Encoder

```
(v: Email) -> preparedStatement(n, v)
```

info@exoquery.io

```
preparedStatement.setParam(1, runtimeName)
```

```
SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?
```

Also, how do I get runtime data *out*?

```
val runtimeName: String = somethingFromSomewhere()

val query: SqlQuery<Person> =
  capture {
    Table<Person>().filter { p -> p.name == 'Joe' }
  }

val people: List<Person> =
  query.buildFor.Postgres().runOn(db)
```


```
[
  Person(id=1, name=Joe, email=joe@someplace.com),
  Person(id=2, name=Joe, email=joer@someplace.com)
]
```

Also, how do I get runtime data *out*?

```
val runtimeName: String = somethingFromSomewhere()

val query: SqlQuery<Person> =
  capture {
    Table<Person>().filter { p -> p.name == 'Joe' }
  }

val people: List<Person> =
  query.buildFor.Postgres().runOn(db)
```


```
@Serializable
data class Person(val id: Int, val name: String, val email: String)
```

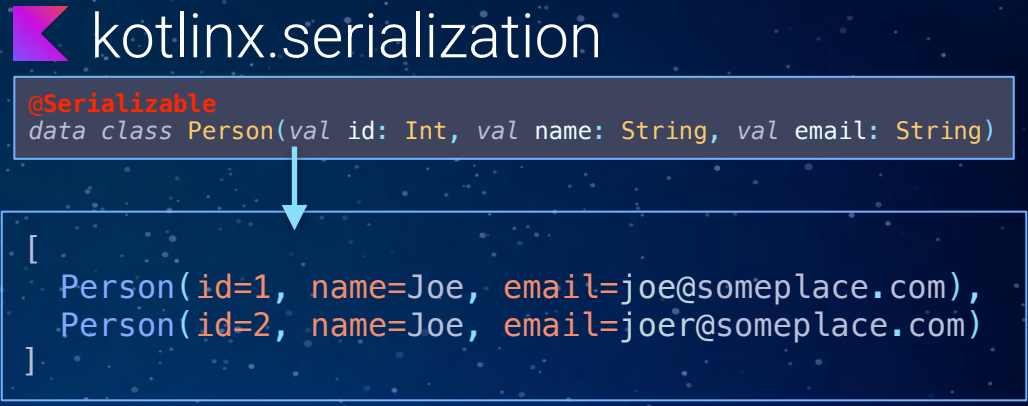
```
[
  Person(id=1, name=Joe, email=joe@someplace.com),
  Person(id=2, name=Joe, email=joer@someplace.com)
]
```

Also, how do I get runtime data *out*?

```
val runtimeName: String = somethingFromSomewhere()

val query: SqlQuery<Person> =
  capture {
    Table<Person>().filter { p -> p.name == 'Joe' }
  }

val people: List<Person> =
  query.buildFor.Postgres().runOn(db)
```

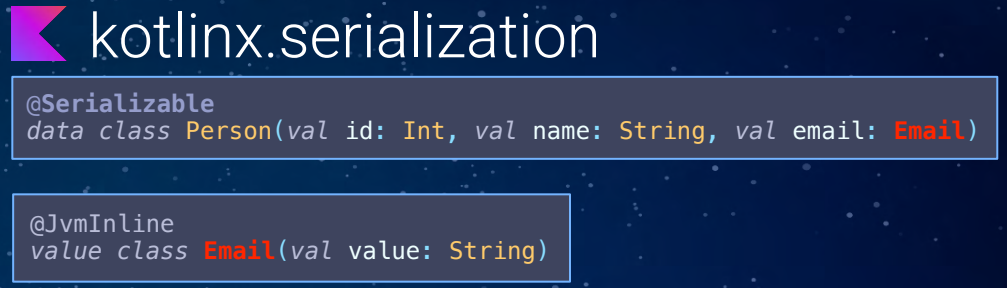


Also, how do I get runtime data *out*?

```
val runtimeName: String = somethingFromSomewhere()

val query: SqlQuery<Person> =
  capture {
    Table<Person>().filter { p -> p.name == 'Joe' }
  }

val people: List<Person> =
  query.buildFor.Postgres().runOn(db)
```



Also, how do I get runtime data *out*?

```
val runtimeName: String = somethingFromSomewhere()

val query: SqlQuery<Person> =
  capture {
    Table<Person>().filter { p -> p.name == 'Joe' }
  }

val people: List<Person> =
  query.buildFor.Postgres().runOn(db)
```

kotlinx.serialization

Have custom fields?
User defined nested types?
Use kotlinx.serialization custom serializers

```
@Serializable
data class Person(val id: Int, val name: String,
  @Serializable(with = EmailSerializer::class)
  val email: Email
)
```

```
@ExoValue
@JvmInline
value class Email(val value: String)
```

```
[
  Person(id=1, name=Joe, email=Email(value=joe@someplace.com)),
  Person(id=2, name=Joe, email=Email(value=joer@someplace.com))
]
```

ExoQuery
ADVANCED TOPIC

`kotlin.serialization` ~= Generic Derivation

`kotlin.serialization` ~= **Generic Derivation**

Why are we doing it like this?

`kotlin.serialization` ~= **Generic Derivation**

Why are we doing it like this? ~~**Reflection**~~

`kotlin.serialization` ~= **Generic Derivation**

Why are we doing it like this? ~~**Reflection**~~

If ~~**Reflection**~~ 

`kotlin.serialization` ~= **Generic Derivation**

Why are we doing it like this? ~~**Reflection**~~

If ~~**Reflection**~~ ⟶  Kotlin Multiplatform

`kotlin.serialization` ~= **Generic Derivation**

Why are we doing it like this? ~~**Reflection**~~

If ~~**Reflection**~~ ⟶ Kotlin Multiplatform

`kotlin.serialization` ~= **Generic Derivation**

Why are we doing it like this? ~~**Reflection**~~

If ~~**Reflection**~~ → Kotlin Multiplatform

↓

Native Binaries

`kotlin.serialization` ~= **Generic Derivation**

Why are we doing it like this? ~~**Reflection**~~

If ~~**Reflection**~~ → Kotlin Multiplatform

Native Binaries

Remember...
Compile Time Queries -> Zero Runtime Cost
Ideal for Mobile Development

kotlin.serialization ~= Generic Derivation

Why are we doing it like this? ~~Reflection~~

If ~~Reflection~~ → Kotlin Multiplatform

Native Binaries

Remember...
Compile Time Queries -> Zero Runtime Cost
Ideal for Mobile Development
https://github.com/ExoQuery/exoquery-sample-kmp

Actions

(a.k.a. CRUD without tears)

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val q =
  capture {
    insert<Person> { set(name to "Joe", age to 123) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES ('Joe', 123)
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val q =
  capture {
    insert<Person> { set(name:String to "Joe", age:String to 123) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES ('Joe', 123)
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val nameVal = "Joe"
val ageVal = 123

val q =
  capture {
    insert<Person> { set(name to param(nameVal), age to param(ageVal)) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?)
```

Insert

```
val name = "Joe"
val age = 123

val q =
  capture {
    insert<Person> { set(name to param(name), age to param(age)) }
  }

q.build<PostgresDialect>()
```

Insert

```
val name = "Joe"
val age = 123

val q =
  capture {
    insert<Person> { set(name to param(name), age to param(age)) }
  }
```

```
It looks like the variable `name` is coming from outside the capture/expression block. Typically
this is a runtime-value that you need to bring into the query as a parameter like this: `param(name)`.
For example:

val nameVariable = "Joe"
val query = select { Table<Person>().filter { p -> p.name == param(nameVariable) } }
> This will create the query:
> SELECT p.id, p.name, p.age FROM Person p WHERE p.name = ?

(Lineage: [val name->val.Owner]->[fun insertAction(...)->fun.Owner]->[File(io.exoquery.example.Part5_Actions.kt)]).
------------ Source ------------
name
------------ Raw Expression ------------
name
------------ Raw Expression Tree ------------
[IrGetValue] Var(name) (orig=null) (param=DEFINED)
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val nameVal = "Joe"
val ageVal = 123

val q =
  capture {
    insert<Person> { set(name to param(nameVal), age to param(ageVal)) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?)
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val nameVal = "Joe"
val ageVal = 123

val q =
  capture {
    insert<Person> { set(name to param(nameVal), age to param(ageVal)) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?)
```

Do I really need to write out ever parameter???

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    insert<Person> { setParams(p) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (id, name, age) VALUES (?, ?, ?)
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    insert<Person> { setParams(p) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (id, name, age) VALUES (?, ?, ?)
```

No! Don't insert an id?

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?)
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    insert<Person> { setParams(p).excluding(id, otherStuff...) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?)
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

with Returning

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    insert<Person> { setParams(p).excluding(id) }.returning { p -> p.id }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?) RETURNING id
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

with Returning

```
val p = Person(1, "Joe", 123)

val q:SqlAction<Person, Int> =
  capture {
    insert<Person> { setParams(p).excluding(id) }.returning { p -> p.id }
  }

val id:Int = q.build<PostgresDialect>().runOn(db)
```

```
INSERT INTO Person (name, age) VALUES (?, ?) RETURNING id
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
@Ser data class Output(val id: Int, val name: String)
```

```
val p = Person(1, "Joe", 123)

val q:SqlAction<Person, Output> =
  capture {
    insert<Person> { setParams(p).excluding(id) }.returning { p -> Output(p.id, p.name) }
  }

val id:Output = q.build<PostgresDialect>().runOn(db)
```

```
INSERT INTO Person (name, age) VALUES (?, ?) RETURNING id, name
```

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
@Ser data class Output(val id: Int, val name: String)
```

```
val p = Person(1, "Joe", "joe@someplace.com")

val q:SqlAction<Person, Output> =
  capture {
    insert<Person> { setParams(p).excluding(id) }
      .returning { p -> Output(p.id, p.name + "-" + p.email) }
  }

val id:Output = q.build<PostgresDialect>().runOn(db)
```

```
INSERT INTO Person (name, age) VALUES (?, ?) RETURNING id, (name || '-') || email
```

ExoQuery
ADVANCED TOPIC
info@exoquery.io

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
@Ser data class Output(val id: Int, val name: String)
```

```
val p = Person(1, "Joe", "joe@someplace.com")

val q:SqlAction<Person, Output> =
  capture {
    insert<Person> { setParams(p).excluding(id) }
      .returning { p -> Output(p.id, p.name + "-" + p.email) }
  }

val id:Output = q.build<PostgresDialect>().runOn(db)
```

```
INSERT INTO Person (name, age) VALUES (?, ?) RETURNING id, (name || '-') || email
```

Not Universally Supported

Insert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)

val q:SqlAction<Person, Int> =
  capture {
    insert<Person> { setParams(p).excluding(id) }.returningKeys { id }
  }

val id:Int = q.build<PostgresDialect>().runOn(db)
```

```
INSERT INTO Person (name, age) VALUES (?, ?) RETURNING id, name
```

Uses preparedStatement.generatedKeys

ExoQuery
ADVANCED TOPIC
info@exoquery.io

Upsert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

OnConflict Update

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    insert<Person> {
      setParams(p).onConflictUpdate(id) { excluding ->
        set(name to excluding.name)
      }
    }
  }

q.build<PostgresDialect>().runOn(db)
```

```
INSERT INTO Person AS x (id, name, age) VALUES (?, ?, ?)
ON CONFLICT (id) DO UPDATE SET name = EXCLUDED.name
```

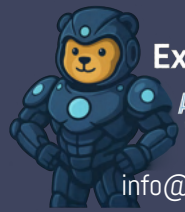
Upsert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    insert<Person> {
      setParams(p).onConflictUpdate(id) { excluding ->
        set(name to name + "-" + excluding.name)
      }
    }
  }

q.build<PostgresDialect>().runOn(db)
```



ExoQuery
ADVANCED
TOPIC
info@exoquery.io

```
INSERT INTO Person AS x (id, name, age) VALUES (?, ?, ?)
ON CONFLICT (id) DO UPDATE SET name = ((x.name || '-') || EXCLUDED.name)
```

Upsert

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

OnConflict Do Nothing

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    insert<Person> {
      setParams(p).onConflictIgnore(id)
    }
  }

q.build<PostgresDialect>().runOn(db)
```

```
INSERT INTO Person AS x (id, name, age) VALUES (?, ?, ?)
ON CONFLICT (id) DO NOTHING
```

Update

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val nameVal = "Joe"
val ageVal = 123
val someParam = 123

val q =
  capture {
    update<Person> { set(name to param(nameVal), age to param(ageVal)) }
      .where { id == param(someParam) }
  }


q.build<PostgresDialect>()
```

```
UPDATE Person SET name = ?, age = ? WHERE id = ?
```

Update

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)
```

Update has Same Syntax

```
val q =
  capture {
    update<Person> { setParams(p).excluding(id) }
      .where { id == param(p.id) }
  }


q.build<PostgresDialect>()
```

```
UPDATE Person SET name = ?, age = ? WHERE id = ?
```

Update

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)


val q =
  capture {
    update<Person> { setParams(p).excluding(id) }.returning(...)
      .where { id == param(p.id) }
  }


q.build<PostgresDialect>()
```

```
UPDATE Person SET name = ?, age = ? WHERE id = ? RETURNING ...
```

Update

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

Want no 'WHERE' clause? Specify it explicitly!

```
val p = Person(1, "Joe", 123)



val q =
  capture {
    update<Person> { setParams(p).excluding(id) }
      .all()
  }


q.build<PostgresDialect>()
```

```
UPDATE Person SET name = ?, age = ?
```

Delete

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    delete<Person>().where { id == param(p.id) }
  }

q.build<PostgresDialect>()
```

```
DELETE FROM Person WHERE id = ?
```

Delete

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    delete<Person>().where { id == param(p.id) }.returning(...)
  }

q.build<PostgresDialect>()
```

```
DELETE FROM Person WHERE id = ? RETURNING ...
```

Delete

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

Want no 'WHERE' clause? Specify it explicitly!

```
val p = Person(1, "Joe", 123)

val q =
  capture {
    delete<Person>().all()
  }

q.build<PostgresDialect>()
```

```
DELETE FROM Person
```

Batch

```
@Ser data class Person(val id: Int, val name: String, val age: Int)
```

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?)
```

Uses JDBC addBatch/executeBatch

Batch

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id).onConflictUpdate(...) { ... } }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?) ON CONFLICT ...
```

Uses JDBC addBatch/executeBatch

Batch

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }.returning(...)
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?) RETURNING ...
```

Uses JDBC addBatch/executeBatch

Batch

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert/update/delete<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?) RETURNING ...
```

Uses JDBC addBatch/executeBatch

Batch

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?) RETURNING ...
```

Uses JDBC addBatch/executeBatch

Can we do better than this?

Coming Soon: Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?), (?, ?), ...
```

Coming Soon: Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?), (?, ?), ... Placeholder Limit!
```

Coming Soon: Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

```
INSERT INTO Person (name, age) VALUES (?, ?), (?, ?), ... 65535/2
```

Coming Soon: Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

Placeholder Limits!

```
INSERT INTO Person (name, age) VALUES (?, ?), (?, ?), ... 32767/2
```

...

Placeholder Limits!

```
INSERT INTO Person (name, age) VALUES (?, ?), (?, ?), ... 32767/2
```

...

Done!

Coming Soon: Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

Placeholder Limits!

```
INSERT INTO Person (name, age) VALUES (?, ?), (?, ?), ... 2100/2
```

...

Placeholder Limits!

```
INSERT INTO Person (name, age) VALUES (?, ?), (?, ?), ... 2100/2
```

...

Done!

Coming Soon: Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

Direct Inlining? What about SQL Injection?

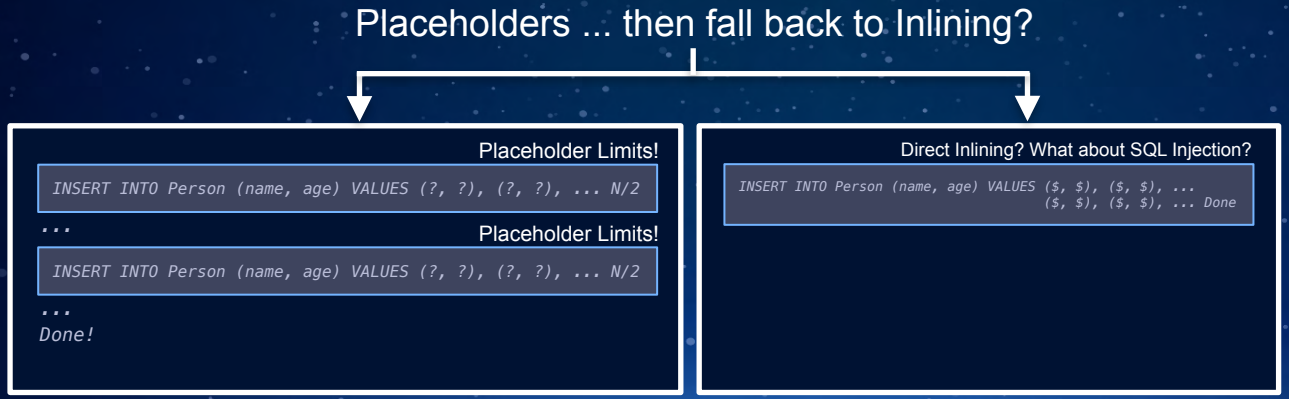
```
INSERT INTO Person (name, age) VALUES ($, $), ($, $), ...
                                      ($, $), ($, $), ... Done
```

Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

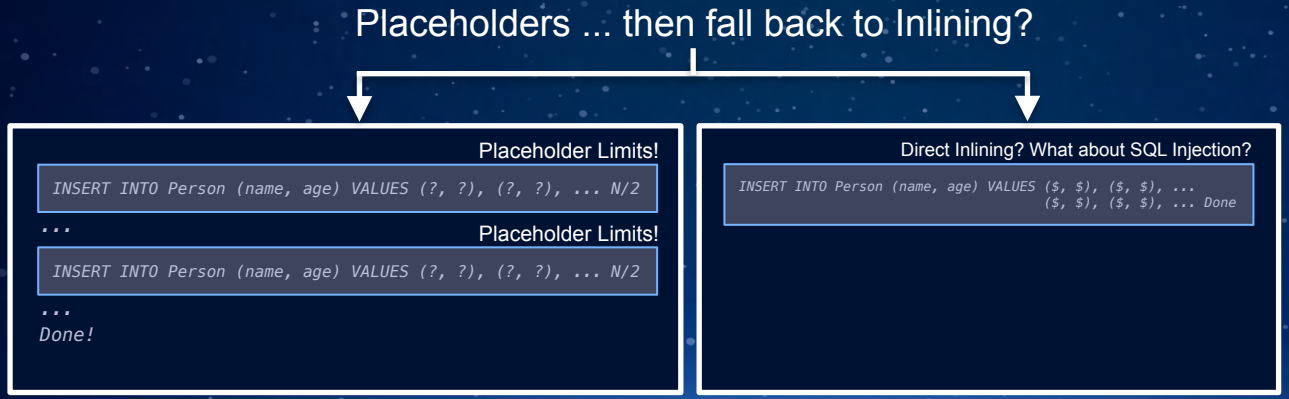


Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batch(people) { p ->
    insert<Person> { setParamsInline(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

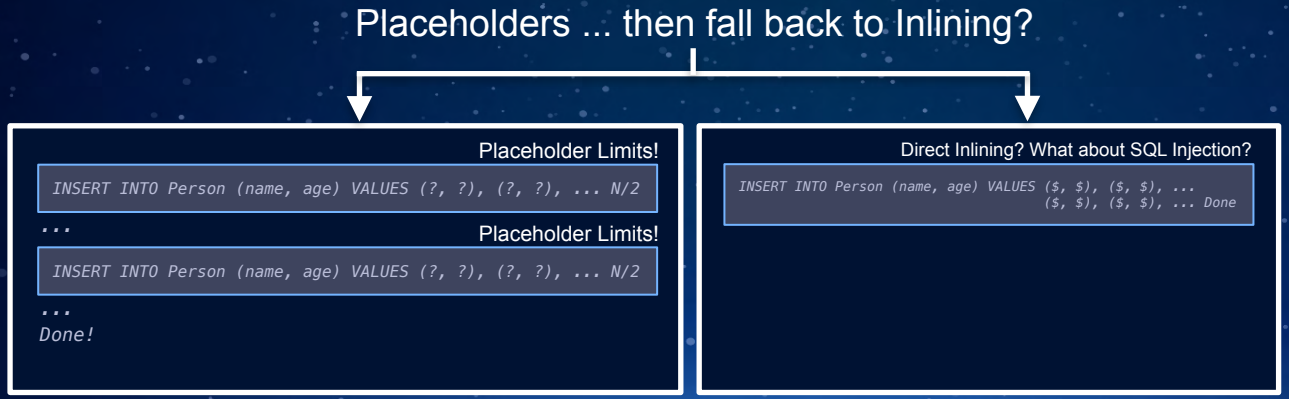


Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batchAuto(people) { p ->
    insert<Person> { setParams(p).excluding(id) }
  }

q.build<PostgresDialect>()
```

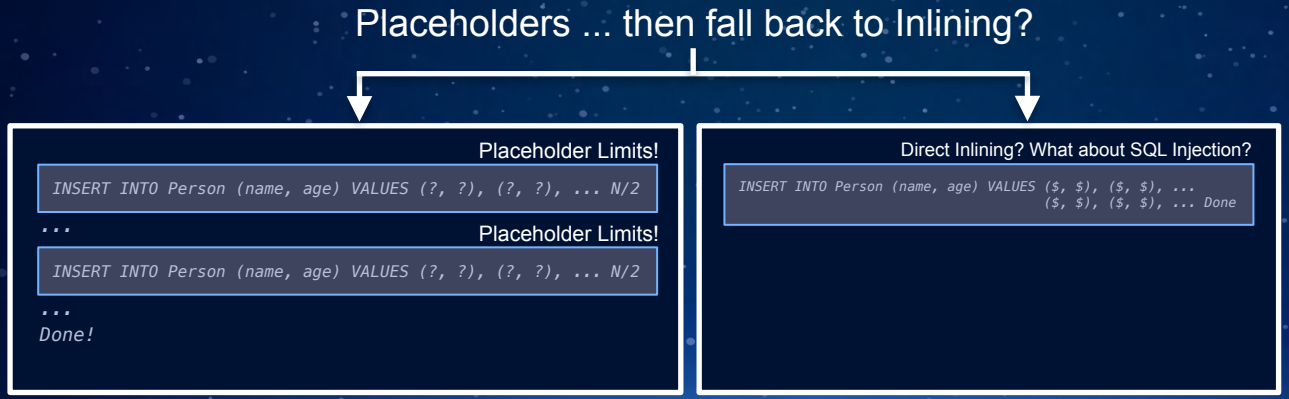


Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batchAuto(people) { p ->
    insert<Person> { setParamsInline(p).excluding(id).onConflict }
      .returning
  }

q.build<PostgresDialect>()
```



Batch Concatenation

```
val people: Sequence<Person> = ...

val q =
  capture.batchAuto(people) { p ->
    insert<Person> { setParamsInline(p).excluding(id).onConflict }
      .returning
  }

q.build<PostgresDialect>()
```

Placeholders ... then fall back to Inlining?

Placeholder Limits!

```
INSERT INTO Person (name, age) VALUES (?, ?), (?, ?), ... N/2
```

...

Placeholder Limits!

```
INSERT INTO Person (name, age) VALUES (?, ?), (?, ?), ... N/2
```

...

Done!

Direct Inlining? What about SQL Injection?

```
INSERT INTO Person (name, age) VALUES ($, $), ($, $), ...
                                      ($, $), ($, $), ... Done
```

Pro Features Planned

- Batch Concatenation + Multiplatform Batch Support
- Inline Parameters and Dynamic Columns
- Advanced query flattening optimization (very fast unions!)
- INSERT INTO ... SELECT
- UPDATE with JOIN
- Schema DDL
- Database from Query
- Auto Query Performance Tracking
- Advanced Json Operators

Open Source Features Planned

- JS Support
- Oracle Support
- @NamingConvention annotations
- Spring Integration
- Hibernate Integration
- Dynamic Query Caching
- @Id annotation with Auto-Exclusion
- Coproduct Rows
- Json-Valued Columns, Basic Operators
- Data Class Entity Generators
- File-Level Query Control
- Much more!

The Grand Finale

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

Remember this guy?

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
     INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```kotlin
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
    val alias: String,
    val code: String,
    val permission: String,
    val tag: String
)
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```kotlin
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
    val alias: St
    val code: Str
    val permissio
    val tag: Stri
)
```

```kotlin
@ExoEntity("SERVICE_CLIENT")
data class ServiceClients(
  val alias: String,
  val partnershipFk: Int,
  val accountTag: String
)
```

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
    val alias: St
    val code: Str
    val permissio
    val tag: Stri
)
```

```
@ExoEntity("SERVICE_CLIENT")
data class ServiceClients(
  val alias: Strin
  val partnershi
  val accountTag
)
```

```
data class Client(
  val alias: String,
  val code: String,
  val permission: String,
  val tag: String
)
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```kotlin
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
    val alias: St
    val code: Str
    val permissio
    val tag: Stri
)
```

```kotlin
@ExoEntity("SERVICE_CLIENT")
data class ServiceClients(
  val alias: Strin
  val partnershi
  val accountTag
)
```

```kotlin
data class Client(
  val alias: String,
  val code: String,
  val permission: String,
  val tag: String
)
```

```kotlin
data class Accounts(
  val name: String,
  val tag: String,
  val number: Int,
  val type: String
)
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
    val alias: St
    val code: Str
    val permissio
    val tag: Stri
)
```

```
@ExoEntity("SERVICE_CLIENT")
data class ServiceClients(
  val alias: Strin
  val partnershi
  val accountTag
)
```

```
data class Client(
  val alias: String,
  val code: String,
  val permission: String,
  val tag: String
)
```

```
data class Accounts(
  val name: String,
  val t
  val n
  val t
)
```

```
@ExoEntity("ACCOUNT_TYPES")
data class AccountTypes(
  val accountType: String,
  val mappingType: Int
)
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
    val alias: St
    val code: Str
    val permissio
    val tag: Stri
)
```

```
@ExoEntity("SERVICE_CLIENT")
data class ServiceClients(
  val alias: Strin
  val partnershi
  val accountTag
)
```

```
data class Client(
  val alias: String,
  val code: String,
  val permission: String,
  val tag: String
)
```

```
data class Accounts(
  val name: String,
  val t
  val n
  val t
)
```

```
@ExoEntity("ACCOUNT_TYPES")
data c
  val
  val
)
```

```
@ExoEntity("ORDER_PERMISSION_TYPES")
data class OrderPermissionTypes(
  val type: String,
  val description: String
)
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```kotlin
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
    val alias: St
    val code: Str
    val permissio
    val tag: Stri
)
```

```kotlin
@ExoEntity("SERVICE_CLIENT")
data class ServiceClients(
  val alias: String
  val partnershi
  val accountTag
)
```

```kotlin
data class Client(
  val alias: String,
  val code: String,
  val permission: String,
  val tag: String
)
```

```kotlin
data class Accounts(
  val name: String,
  val t
  val n
  val t
)
```

```kotlin
@ExoEntity("ACCOUNT_TYPES")
data c
  val
  val
)
```

```kotlin
@ExoEntity("ORDER_PERMISSION_TYPES")
data class OrderPermissionTypes(
  val t
  val c
)
```

```kotlin
data class Partnerships(
  val id: Int,
  val orderPermission: String,
  val description: String
)
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
    val alias: St
    val code: Str
    val permissio
    val tag: Stri
)
```

```
@ExoEntity("SERVICE_CLIENT")
data class ServiceClients(
  val alias: Strin
  val partnershi
  val accountTag
)
```

```
data class Client(
  val alias: String,
  val code: String,
  val permission: String,
  val tag: String
)
```

```
data class Accounts(
  val name: String,
  val t
  val n
  val t
)
```

```
@ExoEntity("ACCOUNT_TYPES")
data c
  val
  val
)
```

```
@ExoEntity("ORDER_PERMISSION_TYPES")
data class OrderPermissionTypes(
  val t
  val c
)
```

```
data class Partnerships(
  val id: Int
  va
  va
)
```

```
@ExoEntity("DEDICATED_ACCOUNTS")
data class DedicatedAccounts(
  val accountNumber: Int,
  val clientAlias: String
)
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
    val alias: St
    val code: Str
    val permissio
    val tag: Stri
)
```

```
@ExoEntity("SERVICE_CLIENT")
data class ServiceClients(
  val alias: Strin
  val partnershi
  val accountTag
)
```

```
data class Client(
  val alias: String,
  val code: String,
  val permission: String,
  val tag: String
)
```

```
data class Accounts(
  val name: String,
  val t
  val n
  val t
)
```

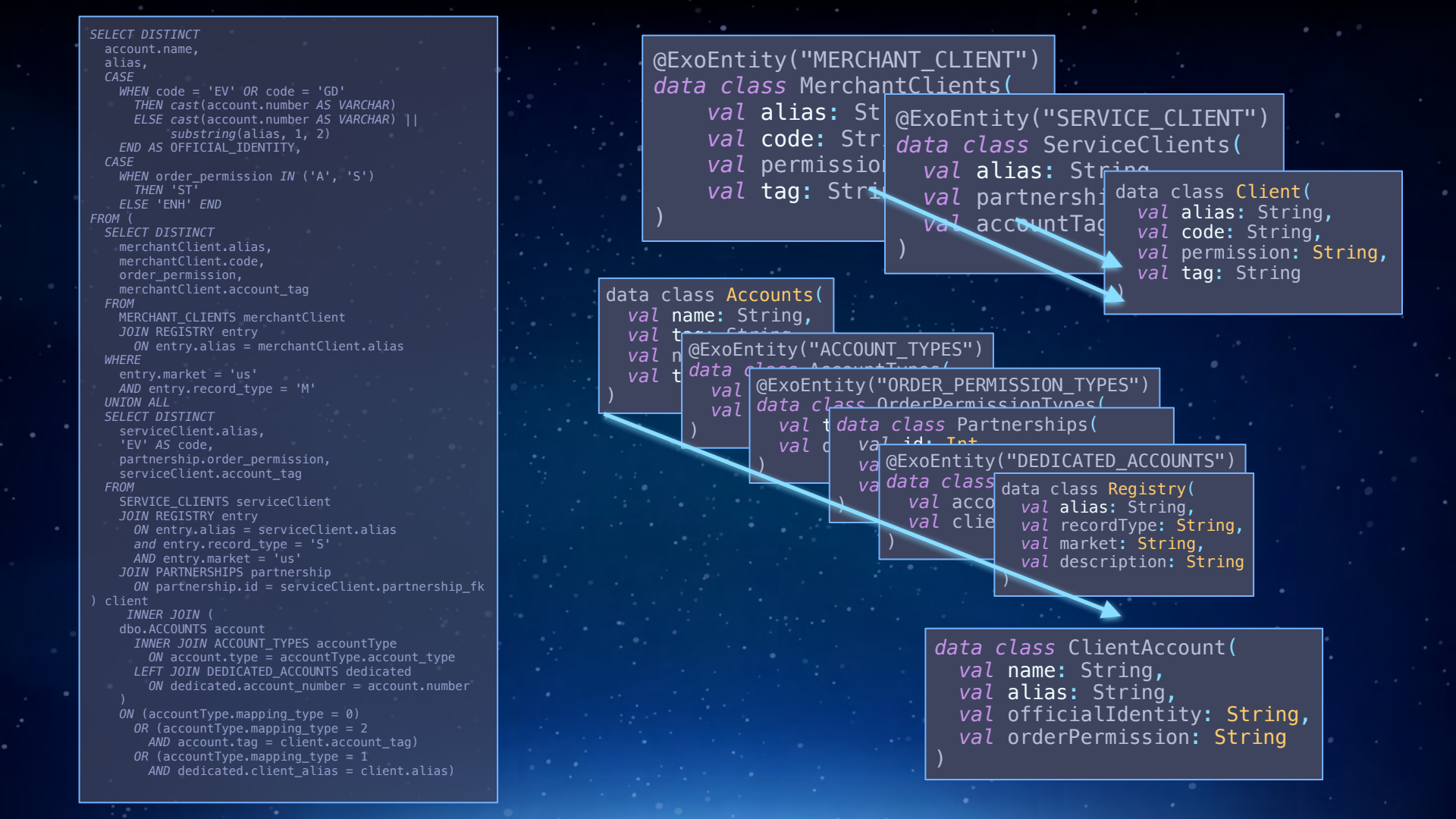
```
@ExoEntity("ACCOUNT_TYPES")
data c
  val
  val
)
```

```
@ExoEntity("ORDER_PERMISSION_TYPES")
data class OrderPermissionTypes(
  val t
  val c
)
```

```
data class Partnerships(
  val id: Int
  va
  va
)
```

```
@ExoEntity("DEDICATED_ACCOUNTS")
data class
  val acco
  val clie
)
```

```
data class Registry(
  val alias: String,
  val recordType: String,
  val market: String,
  val description: String
)
```


SELECT DISTINCT
account.name,
alias,
CASE
WHEN code = 'EV' OR code = 'GD'
THEN cast(account.number AS VARCHAR)
ELSE cast(account.number AS VARCHAR) ||
substring(alias, 1, 2)
END AS OFFICIAL_IDENTITY,
CASE
WHEN order_permission IN ('A', 'S')
THEN 'ST'
ELSE 'ENH' END
FROM (
SELECT DISTINCT
merchantClient.alias,
merchantClient.code,
order_permission,
merchantClient.account_tag
FROM
MERCHANT_CLIENTS merchantClient
JOIN REGISTRY entry
ON entry.alias = merchantClient.alias
WHERE
entry.market = 'us'
AND entry.record_type = 'M'
UNION ALL
SELECT DISTINCT
serviceClient.alias,
'EV' AS code,
partnership.order_permission,
serviceClient.account_tag
FROM
SERVICE_CLIENTS serviceClient
JOIN REGISTRY entry
ON entry.alias = serviceClient.alias
and entry.record_type = 'S'
AND entry.market = 'us'
JOIN PARTNERSHIPS partnership
ON partnership.id = serviceClient.partnership_fk
) client
INNER JOIN (
dbo.ACCOUNTS account
INNER JOIN ACCOUNT_TYPES accountType
ON account.type = accountType.account_type
LEFT JOIN DEDICATED_ACCOUNTS dedicated
ON dedicated.account_number = account.number
)
ON (accountType.mapping_type = 0)
OR (accountType.mapping_type = 2
AND account.tag = client.account_tag)
OR (accountType.mapping_type = 1
AND dedicated.client_alias = client.alias)
@ExoEntity("MERCHANT_CLIENT")
data class MerchantClients(
val alias: St
val code: Str
val permissio
val tag: Stri
)
@ExoEntity("SERVICE_CLIENT")
data class ServiceClients(
val alias: Strin
val partnershi
val accountTag
)
data class Client(
val alias: String,
val code: String,
val permission: String,
val tag: String
)
data class Accounts(
val name: String,
val t
val n
val t
)
@ExoEntity("ACCOUNT_TYPES")
data c
val
val
)
@ExoEntity("ORDER_PERMISSION_TYPES")
data class OrderPermissionTypes(
val t
val c
)
data class Partnerships(
val id: Int
va
va
)
@ExoEntity("DEDICATED_ACCOUNTS")
data class
val acco
val clie
)
data class Registry(
val alias: String,
val recordType: String,
val market: String,
val description: String
)
data class ClientAccount(
val name: String,
val alias: String,
val officialIdentity: String,
val orderPermission: String
)

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
     INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
fun merchantClients() =
  capture.select {
    val merchantClient = from(Table<MerchantClients>())
    val entry = join(Table<Registry>()) { entry -> merchantClient.alias == entry.alias }
    where { entry.market == "us" && entry.recordType == "M" }
    Client(
      merchantClient.alias,
      merchantClient.code,
      merchantClient.orderPermission,
      merchantClient.accountTag
    )
  }
```

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
fun merchantClients() =
  capture.select {
    val merchantClient = from(Table<MerchantClients>())
    val entry = join(Table<Registry>()) { entry -> merchantClient.alias == entry.alias }
    where { entry.market == "us" && entry.recordType == "M" }
    Client(
      merchantClient.alias,
  }
```

```
fun serviceClients() =
  capture.select {
    val serviceClient = from(Table<ServiceClients>())
    val entry = join(Table<Registry>()) { entry -> serviceClient.alias == entry.alias }
    val partnership = join(Table<Partnerships>()) { partnership -> partnership.id == serviceClient.partnershipFk }
    where { entry.market == "us" && entry.recordType == "S" }
    Client(
      serviceClient.alias,
      "EV", // hardcoded code as per the original query
      partnership.orderPermission,
      serviceClient.accountTag
    )
  }
```

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
fun merchantClients() =
  capture.select {
    val merchantClient = from(Table<MerchantClients>())
    val entry = join(Table<Registry>()) { entry -> merchantClient.alias == entry.alias }
    where { entry.market == "us" && entry.recordType == "M" }
    Client(
      merchantClient.alias,
  }
```

```
fun serviceClients() =
  capture.select {
    val serviceClient = from(Table<ServiceClients>())
    val entry = join(Table<Registry>()) { entry -> serviceClient.alias == entry.alias }
    val partnership = join(Table<Partnerships>()) { partnership -> partnership.id == serviceClient.partnershipFk }
    where { entry.market == "us" && entry.recordType == "S" }
    Client(
      serviceClient.alias,
      "EV", // hardcoded code as per the original query
      partnership.orderPermission,
      serviceClient.accountTag
    )
  }
```

```
fun allClients(): SqlQuery<Client> = capture {
  serviceClients() union merchantClients()
}
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
fun merchantClients():SqlQuery<Client> =
  capture.select {
    val merchantClient = from(Table<MerchantClients>())
    val entry = join(Table<Registry>()) { entry -> merchantClient.alias == entry.alias }
    where { entry.market == "us" && entry.recordType == "M" }
    Client(
      merchantClient.alias,
  }
```

```
fun serviceClients():SqlQuery<Client> =
  capture.select {
    val serviceClient = from(Table<ServiceClients>())
    val entry = join(Table<Registry>()) { entry -> serviceClient.alias == entry.alias }
    val partnership = join(Table<Partnerships>()) { partnership -> partnership.id == serviceClient.partnershipFk }
    where { entry.market == "us" && entry.recordType == "S" }
    Client(
      serviceClient.alias,
      "EV", // hardcoded code as per the original query
      partnership.orderPermission,
      serviceClient.accountTag
    )
  }
```

```
fun allClients(): SqlQuery<Client> = capture {
  serviceClients() unionAll merchantClients()
}
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
fun merchantClients():SqlQuery<Client> =
  capture.select {
    val merchantClient = from(Table<MerchantClients>())
    val entry = join(Table<Registry>()) { entry -> merchantClient.alias == entry.alias }
    where { entry.market == "us" && entry.recordType == "M" }
    Client(
  )
}
```

```
fun serviceClients():SqlQuery<Client> =
  capture.select {
    val serviceClient = from(Table<ServiceClients>())
    val entry = join(Table<Registry>()) { entry -> serviceClient.alias == entry.alias }
    val partnership = join(Table<Partnerships>()) { partnership -> partnership.id == serviceClient.partnershipFk }
    where { entry.market == "us" && entry.recordType == "S" }
    Client(
      serviceClient.alias,
      "EV", // hardcoded code as per the original query
      partnership.orderPerm
      serviceClient.account
    )
  }
```

```
fun allClients(): SqlQuery<Client> = capture {
  serviceClients() unionAll merchantClients()
}
```

```
@CapturedFunction
fun clientAccounts(clients: SqlQuery<Client>) = capture.select {
  val client = from(clients)
  val (account, accountType, dedicated) = join(
    capture.select {
      val account = from(Table<Accounts>())
      val accountType = join(Table<AccountTypes>()) { at -> account.type == at.accountType }
      val dedicated = joinLeft(Table<DedicatedAccounts>()) { d -> d.accountNumber == account.number }
      Triple(account, accountType, dedicated)
    }) { (account, accountType, dedicated) ->
    (accountType.mappingType == 0) ||
    (accountType.mappingType == 2 && (account.tag == client.tag)) ||
    (accountType.mappingType == 1 && (dedicated?.clientAlias == client.alias))
  }
  ClientAccount(
    account.name,
    client.alias,
    when (client.code) {
      "EV" -> account.number as String
      else -> (account.number as String) + client.alias.substring(1, 2)
    },
    when (client.permission) {
      "A", "S" -> "ST"
      else -> "ENH"
    }
  )
```

```sql
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
fun merchantClients(): SqlQuery<Client> =
  capture.select {
    val merchantClient = from(Table<MerchantClients>())
    val entry = join(Table<Registry>()) { entry -> merchantClient.alias == entry.alias }
    where { entry.market == "us" && entry.recordType == "M" }
    Client(
```

```
fun serviceClients(): SqlQuery<Client> =
  capture.select {
    val serviceClient = from(Table<ServiceClients>())
    val entry = join(Table<Registry>()) { entry -> serviceClient.alias == entry.alias }
    val partnership = join(Table<Partnerships>()) { partnership -> partnership.id == serviceClient.partnershipFk }
    where { entry.market == "us" && entry.recordType == "S" }
    Client(
      serviceClient.alias,
      "EV", // hardcoded code as per the original query
      partnership.orderPerm
      serviceClient.account
    )
  }
```

```
fun allClients(): SqlQuery<Client> = capture {
  serviceClients() unionAll merchantClients()
}
```

```
@CapturedFunction
fun clientAccounts(clients: SqlQuery<Client>) = capture.select {
  val client = from(clients)
  val (account, accountType, dedicated) = join(
    capture.select {
      val account = from(Table<Accounts>())
      val accountType = join(Table<AccountTypes>()) { at -> account.type == at.accountType }
      val dedicated = joinLeft(Table<DedicatedAccounts>()) { d -> d.accountNumber == account.number }
      Triple(account, accountType, dedicated)
    }) { (account, accountType, dedicated) ->
    (accountType.mappingType == 0) ||
    (accountType.mappingType == 2 && (account.tag == client.tag)) ||
    (accountType.mappingType == 1 && (dedicated?.clientAlias == client.alias))
  }
  ClientAccount(
    account.name,
    client.alias,
    when (client.code) {
      "EV" -> account.number as String
      else -> (account.number as String) + client.alias.substring(1, 2)
    },
    when (client.permission) {
      "A", "S"
      else -> "
    }
  )
```

```
val query = capture {
  clientAccounts(allClients().nested()).distinct()
}
query.buildPretty<PostgresDialect>().value
```

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
SELECT
  d.first_name AS name,
  client.alias,
  CASE
    WHEN client.code = 'EV' THEN d.first_number
    ELSE d.first_number || SUBSTRING(client.alias, 1, 2)
  END AS officialIdentity,
  CASE
    WHEN client.permission = 'A'
    OR client.permission = 'S' THEN 'ST'
    ELSE 'ENH'
  END AS orderPermission
FROM
  (
    (
      SELECT
        merchantClient.alias,
        merchantClient.code,
        merchantClient.permission,
        merchantClient.tag
      FROM
        MerchantClients merchantClient
        INNER JOIN Registry entry ON merchantClient.alias = entry.alias
      WHERE
        entry.market = 'us'
        AND entry.recordType = 'M'
    )
    UNION ALL
    (
      SELECT
        serviceClient.alias,
        'EV' AS code,
        partnership.orderPermission AS permission,
        serviceClient.accountTag AS tag
      FROM
        ServiceClients serviceClient
        INNER JOIN Registry entry1 ON serviceClient.alias = entry1.alias
        INNER JOIN Partnerships partnership ON partnership.id = serviceClient.partnershipFk
      WHERE
        entry1.market = 'us'
        AND entry1.recordType = 'S'
    )
  ) AS client
  INNER JOIN (
    SELECT
      account.name AS first_name,
      account.tag AS first_tag,
      account.number AS first_number,
      account.type AS first_type,
      at.accountType AS second_accountType,
      at.mappingType AS second_mappingType,
      d.accountNumber AS third_accountNumber,
      d.clientAlias AS third_clientAlias
    FROM
      Accounts account
      INNER JOIN AccountTypes at ON account.type = at.accountType
      LEFT JOIN DedicatedAccounts d ON d.accountNumber = account.number
  ) AS d ON d.second_mappingType = 0
  OR d.second_mappingType = 2
  AND d.first_tag = client.tag
  OR d.second_mappingType = 1
  AND d.third_clientAlias = client.alias
```

```
val query = capture {
  clientAccounts(allClients().nested()).distinct()
}
query.buildPretty<PostgresDialect>().value
```

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
SELECT DISTINCT
  d.first_name AS name,
  client.alias,
  CASE
    WHEN client.code = 'EV' THEN d.first_number
    ELSE d.first_number || SUBSTRING(client.alias, 1, 2)
  END AS officialIdentity,
  CASE
    WHEN client.permission = 'A'
    OR client.permission = 'S' THEN 'ST'
    ELSE 'ENH'
  END AS orderPermission
FROM
  (
    (
      SELECT
        merchantClient.alias,
        merchantClient.code,
        merchantClient.permission,
        merchantClient.tag
      FROM
        MerchantClients merchantClient
        INNER JOIN Registry entry ON merchantClient.alias = entry.alias
      WHERE
        entry.market = 'us'
        AND entry.recordType = 'M'
    )
    UNION ALL
    (
      SELECT
        serviceClient.alias,
        'EV' AS code,
        partnership.orderPermission AS permission,
        serviceClient.accountTag AS tag
      FROM
        ServiceClients serviceClient
        INNER JOIN Registry entry1 ON serviceClient.alias = entry1.alias
        INNER JOIN Partnerships partnership ON partnership.id = serviceClient.partnershipFk
      WHERE
        entry1.market = 'us'
        AND entry1.recordType = 'S'
    )
  ) AS client
  INNER JOIN (
    SELECT
      account.name AS first_name,
      account.tag AS first_tag,
      account.number AS first_number,
      account.type AS first_type,
      at.accountType AS second_accountType,
      at.mappingType AS second_mappingType,
      d.accountNumber AS third_accountNumber,
      d.clientAlias AS third_clientAlias
    FROM
      Accounts account
      INNER JOIN AccountTypes at ON account.type = at.accountType
      LEFT JOIN DedicatedAccounts d ON d.accountNumber = account.number
  ) AS d ON d.second_mappingType = 0
  OR d.second_mappingType = 2
  AND d.first_tag = client.tag
  OR d.second_mappingType = 1
  AND d.third_clientAlias = client.alias
```

```
val query = capture {
  clientAccounts(allClients().nested()).distinct()
}
query.buildPretty<PostgresDialect>().value
```

```
SELECT DISTINCT
  account.name,
  alias,
  CASE
    WHEN code = 'EV' OR code = 'GD'
      THEN cast(account.number AS VARCHAR)
      ELSE cast(account.number AS VARCHAR) ||
           substring(alias, 1, 2)
    END AS OFFICIAL_IDENTITY,
  CASE
    WHEN order_permission IN ('A', 'S')
      THEN 'ST'
    ELSE 'ENH' END
FROM (
  SELECT DISTINCT
    merchantClient.alias,
    merchantClient.code,
    order_permission,
    merchantClient.account_tag
  FROM
    MERCHANT_CLIENTS merchantClient
    JOIN REGISTRY entry
      ON entry.alias = merchantClient.alias
  WHERE
    entry.market = 'us'
    AND entry.record_type = 'M'
  UNION ALL
  SELECT DISTINCT
    serviceClient.alias,
    'EV' AS code,
    partnership.order_permission,
    serviceClient.account_tag
  FROM
    SERVICE_CLIENTS serviceClient
    JOIN REGISTRY entry
      ON entry.alias = serviceClient.alias
      and entry.record_type = 'S'
      AND entry.market = 'us'
    JOIN PARTNERSHIPS partnership
      ON partnership.id = serviceClient.partnership_fk
) client
    INNER JOIN (
    dbo.ACCOUNTS account
      INNER JOIN ACCOUNT_TYPES accountType
        ON account.type = accountType.account_type
      LEFT JOIN DEDICATED_ACCOUNTS dedicated
        ON dedicated.account_number = account.number
    )
    ON (accountType.mapping_type = 0)
      OR (accountType.mapping_type = 2
        AND account.tag = client.account_tag)
      OR (accountType.mapping_type = 1
        AND dedicated.client_alias = client.alias)
```

```
SELECT DISTINCT
  d.first_name AS name,
  client.alias,
  CASE
    WHEN client.code = 'EV' THEN d.first_number
    ELSE d.first_number || SUBSTRING(client.alias, 1, 2)
  END AS officialIdentity,
  CASE
    WHEN client.permission = 'A'
    OR client.permission = 'S' THEN 'ST'
    ELSE 'ENH'
  END AS orderPermission
FROM
  (
    (
      SELECT
        merchantClient.alias,
        merchantClient.code,
        merchantClient.permission,
        merchantClient.tag
      FROM
        MerchantClients merchantClient
        INNER JOIN Registry entry ON merchantClient.alias = entry.alias
      WHERE
        entry.market = 'us'
        AND entry.recordType = 'M'
    )
    UNION ALL
    (
      SELECT
        serviceClient.alias,
        'EV' AS code,
        partnership.orderPermission AS permission,
        serviceClient.accountTag AS tag
      FROM
        ServiceClients serviceClient
        INNER JOIN Registry entry1 ON serviceClient.alias = entry1.alias
        INNER JOIN Partnerships partnership ON partnership.id = serviceClient.partnershipFk
      WHERE
        entry1.market = 'us'
        AND entry1.recordType = 'S'
    )
  ) AS client
  INNER JOIN (
    SELECT
      account.name AS first_name,
      account.tag AS first_tag,
      account.number AS first_number,
      account.type AS first_type,
      at.accountType AS second_accountType,
      at.mappingType AS second_mappingType,
      d.accountNumber AS third_accountNumber,
      d.clientAlias AS third_clientAlias
    FROM
      Accounts account
      INNER JOIN AccountTypes at ON account.type = at.accountType
      LEFT JOIN DedicatedAccounts d ON d.accountNumber = account.number
  ) AS d ON d.second_mappingType = 0
  OR d.second_mappingType = 2
  AND d.first_tag = client.tag
  OR d.second_mappingType = 1
  AND d.third_clientAlias = client.alias
```

```
val query = capture {
  clientAccounts(allClients().nested()).distinct()
}
query.buildPretty<PostgresDialect>().value
```

Why the .nested?

```
x.alias,
x.officialIdentity,
x.orderPermission
FROM
  (
    (
      SELECT
        d.first_name AS name,
        unused.alias,
        CASE
          WHEN unused.code = 'EV' THEN d.first_number
          ELSE d.first_number || SUBSTRING(unused.alias, 1, 2)
        END AS officialIdentity,
        CASE
          WHEN unused.permission = 'A'
          OR unused.permission = 'S' THEN 'ST'
          ELSE 'ENH'
        END AS orderPermission
      FROM
        (
          SELECT
            merchantClient.alias,
            merchantClient.code,
            merchantClient.permission,
            merchantClient.tag
          FROM
            MerchantClients merchantClient
            INNER JOIN Registry entry ON merchantClient.alias = entry.alias
          WHERE
            entry.market = 'us'
            AND entry.recordType = 'M'
        ) AS unused
        INNER JOIN (
          SELECT
            account.name AS first_name,
            account.tag AS first_tag,
            account.number AS first_number,
            account.type AS first_type,
            at.accountType AS second_accountType,
            at.mappingType AS second_mappingType,
            d.accountNumber AS third_accountNumber,
            d.clientAlias AS third_clientAlias
          FROM
            Accounts account
            INNER JOIN AccountTypes at ON account.type = at.accountType
            LEFT JOIN DedicatedAccounts d ON d.accountNumber = account.number
        ) AS d ON d.second_mappingType = 0
        OR d.second_mappingType = 2
        AND d.first_tag = unused.tag
        OR d.second_mappingType = 1
        AND d.third_clientAlias = unused.alias
    )
    UNION ALL
    (
      SELECT
        d.first_name AS name,
        unused.alias,
        CASE
          WHEN unused.code = 'EV' THEN d.first_number
          ELSE d.first_number || SUBSTRING(unused.alias, 1, 2)
        END AS officialIdentity,
        CASE
          WHEN unused.permission = 'A'
          OR unused.permission = 'S' THEN 'ST'
          ELSE 'ENH'
        END AS orderPermission
      FROM
        (
          SELECT
            serviceClient.alias,
            'EV' AS code,
            partnership.orderPermission AS permission,
            serviceClient.accountTag AS tag
          FROM
            ServiceClients serviceClient
            INNER JOIN Registry entry1 ON serviceClient.alias = entry1.alias
            INNER JOIN Partnerships partnership ON partnership.id = serviceClient.partnershipFk
          WHERE
            entry1.market = 'us'
            AND entry1.recordType = 'S'
        ) AS unused
        INNER JOIN (
          SELECT
            account.name AS first_name,
            account.tag AS first_tag,
            account.number AS first_number,
            account.type AS first_type,
            at.accountType AS second_accountType,
            at.mappingType AS second_mappingType,
            d.accountNumber AS third_accountNumber,
            d.clientAlias AS third_clientAlias
          FROM
            Accounts account
            INNER JOIN AccountTypes at ON account.type = at.accountType
            LEFT JOIN DedicatedAccounts d ON d.accountNumber = account.number
        ) AS d ON d.second_mappingType = 0
        OR d.second_mappingType = 2
        AND d.first_tag = unused.tag
        OR d.second_mappingType = 1
        AND d.third_clientAlias = unused.alias
    )
  ) AS x
```

```
val query = capture {
  clientAccounts(allClients()).distinct()
}
query.buildPretty<PostgresDialect>().value
```

The Flatter, the Better

Query Compilation Based on the Flattening Transformation

Alexander Ulrich Torsten Grust
Universität Tübingen
Tübingen, Germany
[alexander.ulrich, torsten.grust]@uni-tuebingen.de

ABSTRACT

We demonstrate the insides and outs of a query compiler based on the **flattening transformation**, a translation technique designed by the programming language community to derive efficient data-parallel implementations from iterative programs. Flattening admits the straightforward formulation of intricate query logic—including deeply nested loops over (possibly ordered) data or the construction of rich data structures. To demonstrate the level of expressiveness that

In tandem with established query compilation techniques that we repeatedly draw on here, flattening enables a principled translation that proceeds in small digestible steps—a welcome advance over rather complex and monolithic approaches, including our own [7]. Flattening admits rich user-facing query languages and data models beyond those offered by recent related efforts (*query shredding* [4], for example, lacks support for ordered data, grouping, or aggregation). To make these points we express the examples in

Nested Constructs -> Flat Query -> Better Performance!

clientAccounts(Client)

```
val query = capture {
  clientAccounts(allClients().nested()).distinct()
}
query.buildPretty<PostgresDialect>().value
```

```
val query = capture {
  clientAccounts(allClients().nested()).distinct()
}
query.buildPretty<PostgresDialect>().value
```

clientAccounts(SqlQuery<Client>): SqlQuery<ClientAccount>

Recap

SQL Abstraction is Necessary but Impossible

CTEs, Views, Table-Functions are not valid means of abstraction. A higher level-language is necessary.

No Current Solutions in Kotlin

Kotlin Solutions are Limited at best. Even the functioning ones have DSL-pollution.

Language Integrated Query is Needed

Only with compile-time meta-programming can the right kind of system be constructed.

Recap

SQL Abstraction is Necessary but Impossible

CTEs, Views, Table-Functions are not valid means of abstraction. A higher level-language is necessary.

No Current Solutions in Kotlin

Kotlin Solutions are Limited at best. Even the functioning ones have DSL-pollution.

Language Integrated Query is Needed

Only with compile-time meta-programming can the right kind of system be constructed.

ExoQuery Delivers This

Solution with regular types and no DSL-noise.

Strong and Lawful Foundations

Powerful SELECT operations and Joins

Extended CRUD Action support

Future Professional Tier Offerings

Synthesizes the Toughest Queries with Ease

Thanks!

github.com/exoquery/exoquery

Alexander Ioffe

TRAINING & EXOQUERY PROFESSIONAL

info@exoquery.io