Database project CS-322

Release 0.3

Amos Wenger, Sebastien Zurfluh & Yoan Blanc

June 03, 2012

CONTENTS

CHAPTER

ONE

DELIVERABLE 1

The goal of this deliverable is to design an ER model, a corresponding relational schema and create the database tables in the given database. The organization of the data in files and the given description does not imply neither an ER model nor a relational schema. It is given to help the student understand the format of the data faster. Finally, a discussion about constraints and removing redundant information is expected.

1.1 Create the ER model for the data

1.1.1 The Schema

See the figure on next page.

1.1.2 Decision made building the Entity-Relationship schema

We tried to remove every denormalized fields, meaning fields that are representing information you can obtain using the rest of the data set.

People

- *Players* and *Coaches* are pointing to the same entities the coach's *coach_id* and the player's *ilkid* are defining the same thing (e.g. *MOEDO01*). So we grouped them under the *Person* entity.
- *Player*'s height will be converted into inches, *1ft = 12in*.
- *Player*'s fields like *first_season*, *last_season* or *college* can be obtained from the *Draft* and *Player Season* entities. They are just denormalized fields.

Teams

The teams pretty look like the CSV file, except that we removed the league from it since some teams switched from *ABA* to *NBA* when *ABA* merged into the later.

Conference and leagues

According to Wikipedia, a team belongs to a *Division* which belongs to a *Conference* (being *Eastern* and *Western*). We first linked the team to a *Conference* but the dataset gives the information about the *Conference* only on *All Star* games (where the best players of each *Conferences* create an *All Star* team and play against each other). So, the *Conference* information will only live there because of the dataset.

About *Leagues*, there are two of them *ABA* and *NBA* and a team may have changed during the year 1975 when *ABA* got merged into the *NBA*.

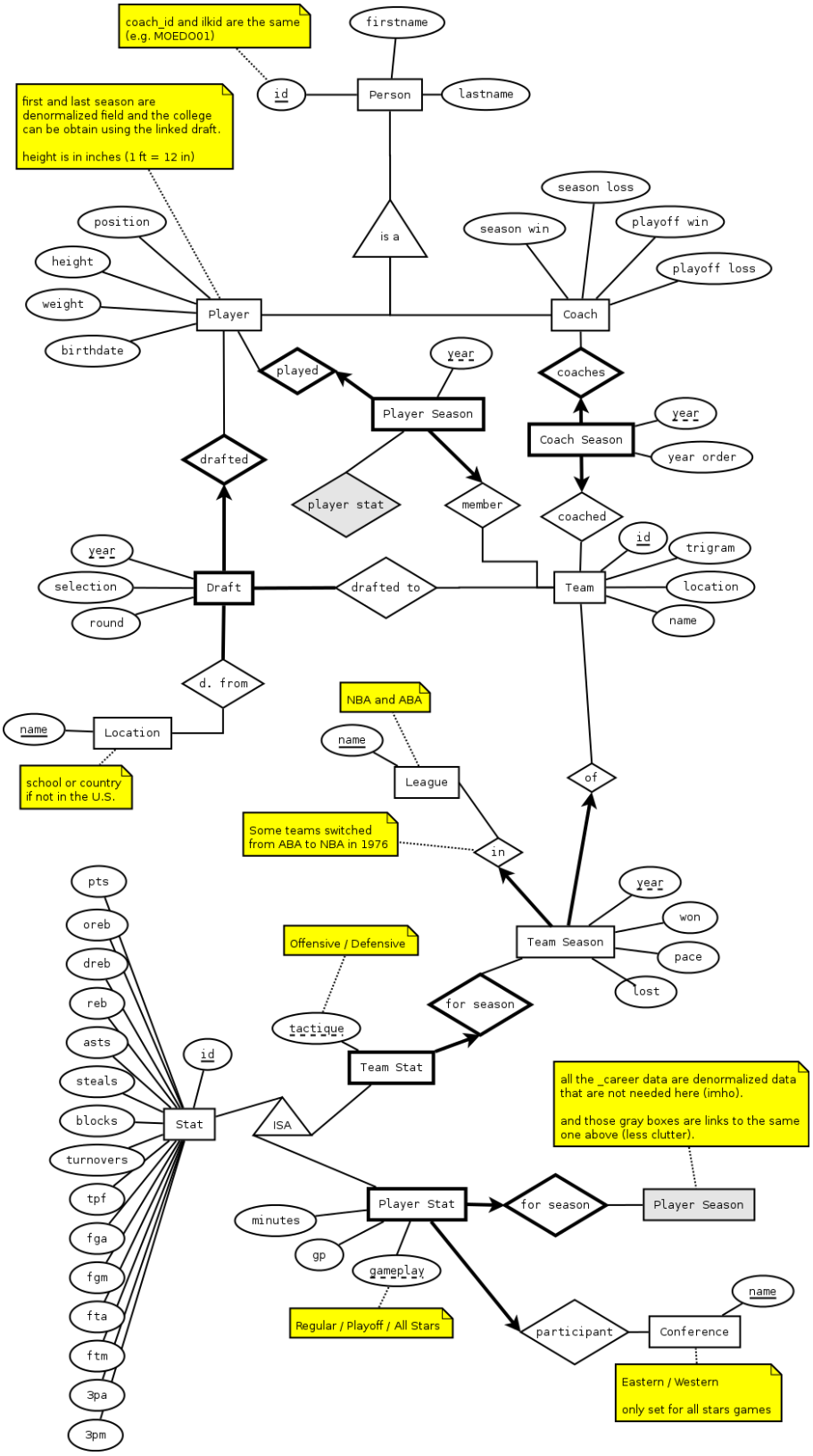


Figure 1.1: ER Schema made using Dia.

Drafts and Location

One major change from the dataset here is that the *People* who go drafted but never played will exist has a *Player*. Those kind of *Player* never played.

Location isn't really important, it's been moved out for further queries.

NB it doesn't have any link with the *Team*'s location which is a city when here, it's the College where the *Player* got drafted from.

Q: *Shouldn't we link Drafts to People and not Players in that case?*

A: No because even players who have never played for an ABA or NBA team have played before for a European team or a school team. Therefore they already have the characteristics of player (position, height, weight, birthdate) when they are drafted.

Stats and Seasons

The *Stats* (statistics) being very standard, it'll live as itself and being kind of *casted* into a *Player* or *Team* stat depending on the case. A *Stat* is all the time linked to a *Season* which is identified by the starting year in the dataset and our model (e.g. *1984* means *the season 84-85*).

Each *Player* and *Team* has a specific *Season* entity linked to it for each *year* played.

Then *Teams* have *offensive* and *defensive* statistics while the *Players* have statistics per kind of *Seasons* played:

- Regular
- Playoff
- and All Stars

NB: All the career stats were seen as denormalized and thus removed. We can get those data back from the yearly *Stats*.

1.2 Design the database and the constraints needed to maintain the database consistent

See the figure on next page.

1.3 Create the SQL commands to create the tables in Oracle

The following SQL schema is really a first shot, with very few constraints on numbers and strings (*varchar*).

Not being familiar with the way Oracle works, we'll just explain some basic stuff.

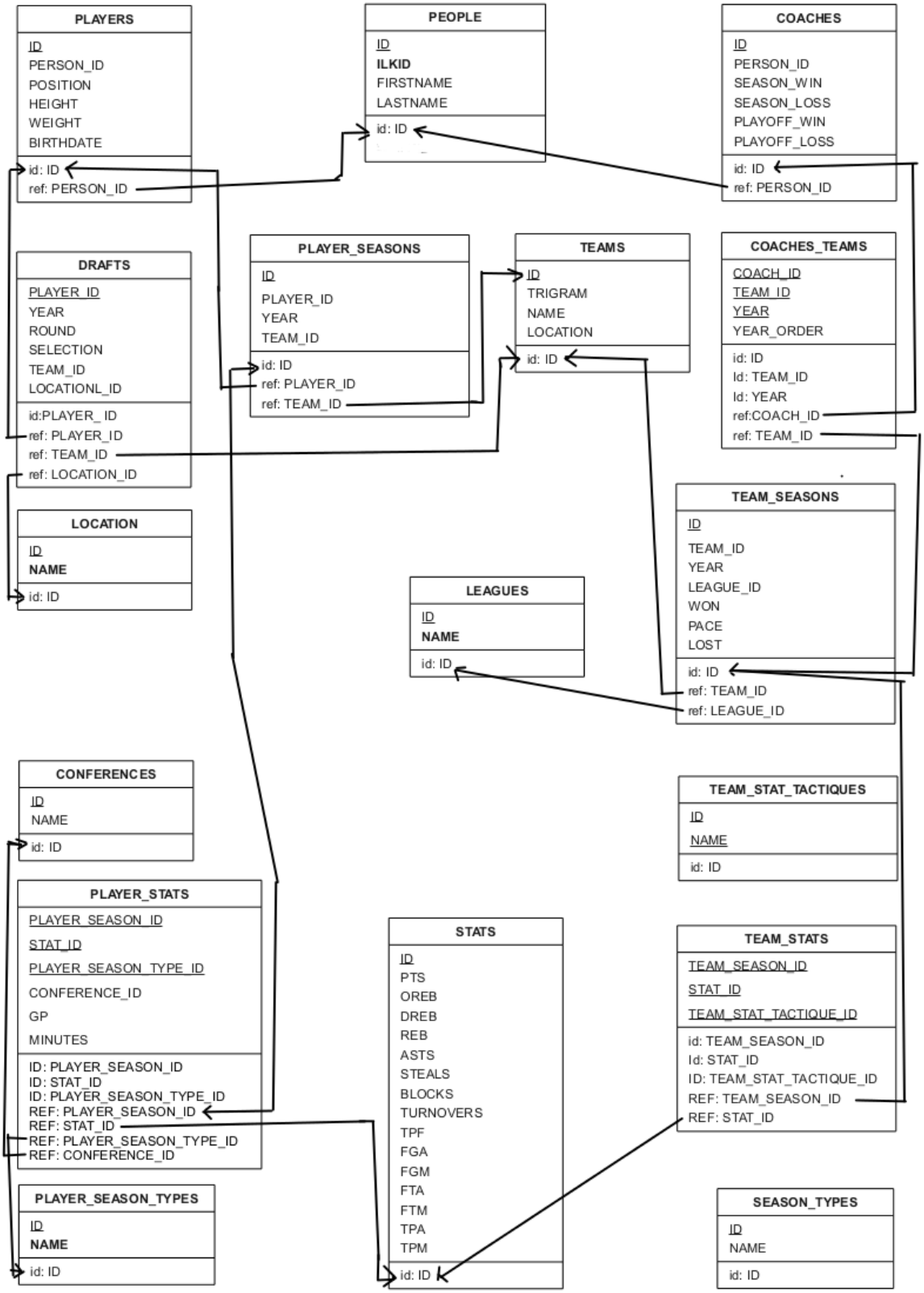
1.3.1 Oracle specificities

Oracle being a very complex RDBMS (*Relational Database Management System*) that we are still learning, this section will simply clarify what we've discovered and wich might explain the following.

It cannot *auto increment* like MySQL or SQLite, so one must use a *sequence* and use it to get the current value or the next value when insert new rows.

```
INSERT INTO leagues (id, name) VALUES (leagues_seq.NEXTVAL, 'NBA');
```

It knows how to delete relations in cascade. It can remove an entire structure from one simple *DELETE*. If *PostgreSQL* can do that has well, the more common *MySQL* or *SQLite* cannot. In the following schema, we activated that cascaded deletion without thinking deeply about it. This schema will be refined in the future when used with the data.


PLAYERS
ID
PERSON_ID
POSITION
HEIGHT
WEIGHT
BIRTHDATE
id: ID
ref: PERSON_ID
PEOPLE
ID
ILKID
FIRSTNAME
LASTNAME
id: ID
COACHES
ID
PERSON_ID
SEASON_WIN
SEASON_LOSS
PLAYOFF_WIN
PLAYOFF_LOSS
id: ID
ref: PERSON_ID
DRAFTS
PLAYER_ID
YEAR
ROUND
SELECTION
TEAM_ID
LOCATIONL_ID
id:PLAYER_ ID
ref: PLAYER_ID
ref: TEAM_ID
ref: LOCATION_ID
PLAYER_SEASONS
ID
PLAYER_ID
YEAR
TEAM_ID
id: ID
ref: PLAYER_ID
ref: TEAM_ID
TEAMS
ID
TRIGRAM
NAME
LOCATION
id: ID
COACHES_TEAMS
COACH_ID
TEAM_ID
YEAR
YEAR_ORDER
id: ID
Id: TEAM_ID
Id: YEAR
ref:COACH_ID
ref: TEAM_ID
LOCATION
ID
NAME
id: ID
TEAM_SEASONS
ID
TEAM_ID
YEAR
LEAGUE_ID
WON
PACE
LOST
id: ID
ref: TEAM_ID
ref: LEAGUE_ID
LEAGUES
ID
NAME
id: ID
CONFERENCES
ID
NAME
id: ID
TEAM_STAT_TACTIQUES
ID
NAME
id: ID
PLAYER_STATS
PLAYER_SEASON_ID
STAT_ID
PLAYER_SEASON_TYPE_ID
CONFERENCE_ID
GP
MINUTES
ID: PLAYER_SEASON_ID
ID: STAT_ID
ID: PLAYER_SEASON_TYPE_ID
REF: PLAYER_SEASON_ID
REF: STAT_ID
REF: PLAYER_SEASON_TYPE_ID
REF: CONFERENCE_ID
STATS
ID
PTS
OREB
DREB
REB
ASTS
STEALS
BLOCKS
TURNOVERS
TPF
FGA
FGM
FTA
FTM
TPA
TPM
id: ID
TEAM_STATS
TEAM_SEASON_ID
STAT_ID
TEAM_STAT_TACTIQUE_ID
id: TEAM_SEASON_ID
Id: STAT_ID
ID: TEAM_STAT_TACTIQUE_ID
REF: TEAM_SEASON_ID
REF: STAT_ID
PLAYER_SEASON_TYPES
ID
NAME
id: ID
SEASON_TYPES
ID
NAME
id: ID

1.3.2 SQL code

```
--
-- People
-- ======
--
-- A person can be a player and/or a coach at different time of
-- her life.
--
-- ilkit can be NULL for drafted only players
--

CREATE TABLE people (
    id NUMBER,
    ilkid VARCHAR(9),
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (ilkid)
);

CREATE SEQUENCE people_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE players (
    id NUMBER,
    person_id NUMBER NOT NULL,
    position CHAR(1) NOT NULL,
    height NUMBER, -- in inches
    weight NUMBER,
    birthdate DATE,
    PRIMARY KEY (id),
    FOREIGN KEY (person_id)
        REFERENCES people (id) ON DELETE CASCADE
);

CREATE SEQUENCE players_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE coaches (
    id NUMBER,
    person_id NUMBER NOT NULL,
    season_win NUMBER,
    season_loss NUMBER,
    playoff_win NUMBER,
    playoff_loss NUMBER,
    PRIMARY KEY (id),
    FOREIGN KEY (person_id)
        REFERENCES people (id) ON DELETE CASCADE
);

CREATE SEQUENCE coaches_seq
    START WITH 1
    INCREMENT BY 1;

--
-- Group of people
-- ===============
--
-- Teams, leagues and stuff
--
```

```
-- NBA/ABA
CREATE TABLE leagues (
    id NUMBER,
    name CHAR(3) NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (name)
);

CREATE SEQUENCE leagues_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE conferences (
    id NUMBER,
    name VARCHAR(31) NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (name)
);

CREATE SEQUENCE conferences_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE teams (
    id NUMBER,
    trigram CHAR(3) NOT NULL,
    name VARCHAR(255),
    location VARCHAR(255),
    PRIMARY KEY (id)
);

CREATE SEQUENCE teams_seq
    START WITH
    INCREMENT BY 1;

CREATE TABLE coaches_teams (
    coach_id NUMBER NOT NULL,
    team_id NUMBER NOT NULL,
    year NUMBER,
    year_order NUMBER,
    PRIMARY KEY (coach_id, team_id, year),
    FOREIGN KEY (coach_id)
        REFERENCES coaches (id) ON DELETE CASCADE,
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE
);

--
-- Physical
-- ========
--
-- A school or a country if it's outside the U.S.
--

CREATE TABLE location (
    id NUMBER,
    name VARCHAR(255),
    PRIMARY KEY (id),
    UNIQUE (name)
);

CREATE SEQUENCE location_seq
```

```
    START WITH 1
    INCREMENT BY 1;

--
-- Drafts
--

CREATE TABLE drafts (
    player_id NUMBER NOT NULL,
    year NUMBER NOT NULL,
    round NUMBER NOT NULL,
    selection NUMBER NOT NULL,
    team_id NUMBER NOT NULL,
    location_id NUMBER NULL,
    PRIMARY KEY (player_id),
    FOREIGN KEY (player_id)
        REFERENCES players (id) ON DELETE CASCADE,
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE,
    FOREIGN KEY (l_id)
        REFERENCES location (id) ON DELETE CASCADE
);

--
-- Stats
-- =====
--
-- All the kind of statistical data
--

CREATE TABLE stats (
    id NUMBER,
    pts NUMBER,
    oreb NUMBER,
    dreb NUMBER,
    reb NUMBER,
    asts NUMBER,
    steals NUMBER,
    blocks NUMBER,
    turnovers NUMBER,
    tpf NUMBER,
    fga NUMBER,
    fgm NUMBER,
    fta NUMBER,
    ftm NUMBER,
    tpa NUMBER, -- 3pa
    tpm NUMBER, -- 3pm
    PRIMARY KEY (id)
);

CREATE SEQUENCE stats_seq
    START WITH 1
    INCREMENT BY 1;

--
-- Teams stats
-- ===========

CREATE TABLE team_seasons (
    id NUMBER,
    team_id NUMBER NOT NULL,
    year NUMBER NOT NULL,
    league_id NUMBER NOT NULL,
```

```
    won NUMBER,
    pace NUMBER,
    lost NUMBER,
    PRIMARY KEY (id),
    CONSTRAINT team_season_unique UNIQUE (team_id, year),
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE,
    FOREIGN KEY (league_id)
        REFERENCES leagues (id) ON DELETE CASCADE
);

CREATE SEQUENCE team_seasons_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE team_stat_tactiques (
    id NUMBER NOT NULL,
    name VARCHAR(31),
    PRIMARY KEY (id),
    UNIQUE (name)
);

CREATE SEQUENCE team_stat_tactiques_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE team_stats (
    team_season_id NUMBER NOT NULL,
    stat_id NUMBER NOT NULL,
    team_stat_tactique_id NUMBER NOT NULL,
    PRIMARY KEY(team_season_id, stat_id, team_stat_tactique_id),
    FOREIGN KEY (team_season_id)
        REFERENCES team_seasons (id) ON DELETE CASCADE,
    FOREIGN KEY (stat_id)
        REFERENCES stats (id) ON DELETE CASCADE
);

--
-- Players stats
-- =============

CREATE TABLE player_seasons (
    id NUMBER,
    player_id NUMBER NOT NULL,
    year NUMBER NOT NULL,
    team_id NUMBER NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT player_season_unique UNIQUE (player_id, year),
    FOREIGN KEY (player_id)
        REFERENCES players (id) ON DELETE CASCADE,
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE
);

CREATE SEQUENCE player_seasons_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE player_season_types (
    id NUMBER,
    name VARCHAR (31),
    PRIMARY KEY (id),
    UNIQUE (name)
```

```
);

CREATE SEQUENCE player_season_types_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE player_stats(
    player_season_id NUMBER NOT NULL,
    stat_id NUMBER NOT NULL,
    player_season_type_id NUMBER NOT NULL,
    conference_id NUMBER NULL, -- for all star games only
    gp NUMBER,
    minutes NUMBER,
    PRIMARY KEY (player_season_id, stat_id, player_season_type_id),
    FOREIGN KEY (player_season_id)
        REFERENCES player_seasons (id) ON DELETE CASCADE,
    FOREIGN KEY (stat_id)
        REFERENCES stats (id) ON DELETE CASCADE,
    FOREIGN KEY (player_season_type_id)
        REFERENCES player_season_types (id) ON DELETE CASCADE,
    FOREIGN KEY (conference_id)
        REFERENCES conferences (id) ON DELETE CASCADE
);
```

1.4 Conclusion

As we decided to go with Ruby on Rails, we expect future changes to be mainly imposed by any limitations that ActiveRecord has.

CHAPTER

TWO

DELIVERABLE 2

The students should accommodate the situation where new data is inserted in any table. Moreover, a simple query which can search for a keyword in any table should be implemented. The user should be able to see more details of the result of the query (e.g., if someone searches for Michael Jordan's regular season statistics and the result has multiple seasons, he/she should be able to see statistics for individual seasons - for example, through a hyperlink).

2.1 Post-mortem deliverable 1

Some of the feedbacks we had.

> *In general, normalizing data is a good thing, but keep in mind that rebuilding some of the information that you discarded might be expensive. Do not hesitate to work with denormalized data if that will make your life easier. That will also help you to have a smaller ER model.*
>
> *On the other hand, working on normalization gave you some great insights on the data. You did a really nice job at studying the domain (i.e., NBA) and exploit such knowledge to fine tune your DB design!*

For the deliverable 2, we didn't had to start denormalizing again but were forced to move data around and denormalizing more in order to gain a better understanding of the dataset. Denormalization adds constraint on data integrity upon changes and we don't what to have to deal with that right now.

But we expect to be forced to start denormalizing fields for the upcoming tasks as things get more complex and intense.

> *[. . .] you did not explain why certain key constraints led to the translation in tables that you will use*

We read your feedbacks on schemas, *ISA* and key constraints but didn't took the time to go back to the design phase. *Getting real* and *hitting walls* mode.

> *Keep up the good work!*

Maybe the only thing we read after all.

2.1.1 Changes in the schema

We made a lot of changes!

NUMBER is not *INT*

The *NUMBER* datatype stores floating numbers. All the *NUMBER* got changed to promper *INT* except for the ones storing real numbers (i.e. *pace*).

ID everywhere

As mentioned in the conclusion of deliverable 1, we expected some changes regarding limitations imposed by the software we are using, ActiveRecord. The first change has been to set up *id* everywhere. The *active record pattern* has trouble working with composed primary keys which is common for tables expressing a *many-to-many* relationship.

```
CREATE TABLE team_stats (
    team_season_id NUMBER NOT NULL,
    stat_id NUMBER NOT NULL,
    team_stat_tactique_id NUMBER NOT NULL,
    PRIMARY KEY(team_season_id, stat_id, team_stat_tactique_id),
    FOREIGN KEY (team_season_id)
        REFERENCES team_seasons (id) ON DELETE CASCADE,
    FOREIGN KEY (stat_id)
        REFERENCES stats (id) ON DELETE CASCADE
);
```

The composite primary key is converted into a unique constraint.

```
CREATE TABLE team_stats (
    id INT,
    team_id INT NOT NULL,
    year INT NOT NULL,
    team_stat_tactique_id INT NOT NULL,
    stat_id INT NOT NULL,
    pace NUMBER NULL,
    PRIMARY KEY (id),
    CONSTRAINT team_stat_unique UNIQUE (team_id, year, team_stat_tactique_id),
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE,
    FOREIGN KEY (team_stat_tactique_id)
        REFERENCES team_stat_tactiques (id) ON DELETE CASCADE,
    FOREIGN KEY (stat_id)
        REFERENCES stats (id) ON DELETE CASCADE
);
```

You can notice that other elements of this table changed. They are explained below.

> *Sidenote:* when using *InnoDB* with MySQL, it's recommended to always have an id because the engine will store the entry on the disk based on that. Specifying it ensure that newest entries are all close to each other. See: http://backchannel.org/blog/friendfeed-schemaless-mysql

Creating *player_allstars*

We initially thought we could aggregate all the *Stat* of one player into a generic table, for the three season types: *Regular*, *Playoff* and *Allstar*. While doing the import, we realized that the later is too different from the others. *Allstar* is **not** linked wia *Team* and thus cannot be tied to a *Player Season* like the two others.

```
CREATE TABLE player_allstars (
    id INT,
    player_id INT NOT NULL,
    stat_id INT NOT NULL,
    conference_id INT NOT NULL,
    year INT NOT NULL,
    gp INT,
    minutes INT,
    PRIMARY KEY(id),
    CONSTRAINT player_allstars_unique UNIQUE (player_id, conference_id, year),
    FOREIGN KEY (player_id)
        REFERENCES players (id) ON DELETE CASCADE,
    FOREIGN KEY (stat_id)
```

```
        REFERENCES stats (id) ON DELETE CASCADE,
    FOREIGN KEY (conference_id)
        REFERENCES conferences (id) ON DELETE CASCADE
);
```

You can notice the same *pattern* describe above with *id* and the unique constraint.

Fixing the *Coaches*

First, we renamed *Coaches Team* to *Coach Season* which makes much more sense.

Secondly, a mistake was made to store. the *season_win*, *season_loss*, ... into the *Coach* itself. This data is the denormalized one and must be into the *Coach Season* instead.

That made us realizing that we had some data duplication. *Team Season* were storing *won*, *pace* and *lost*. *win* and *lost* can be computed using the *Coach Season* and thus got removed from *Team Season*.

```
CREATE TABLE coaches (
    id INT,
    person_id INT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (person_id)
        REFERENCES people (id) ON DELETE CASCADE
);

-- @weak
CREATE TABLE coach_seasons (
    id INT,
    coach_id INT NOT NULL,
    team_id INT NOT NULL,
    year INT NOT NULL,
    year_order INT,
    season_win INT,
    season_loss INT,
    playoff_win INT,
    playoff_loss INT,
    PRIMARY KEY (id),
    CONSTRAINT coach_seasons_unique UNIQUE (coach_id, team_id, year),
    FOREIGN KEY (coach_id)
        REFERENCES coaches (id) ON DELETE CASCADE,
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE
);
```

Moving the *League* where it belongs and dropping *TeamSeason*

We identified *Team* by its *trigram* (those three-letters) after some trouble during the implementation of the import script. The website where our data are coming from (http://http://www.databasebasketball.com/) helped us understanding that it's—in fact—the league identifies the team as well. Knowing that, we moved the *League* from *Team Season* back into *Team* and as the *Team Season* was only containing the information about the *year*, it got moved into the *Team Stat* as well as *pace* which is displayed on the *Offensive* stat on the mother website.

It was also flawed since—to retrieve the league for *Draft*—we had to guess the *Team Season* based on the *Team* and *year* information. And no guarantees are made that that season has (already) been played.

To summarize this change:

- *Team* know which league it belongs to;
- *TeamStats* are referencing a *Team* and contains the information about *year* and *pace*.
- *TeamSeason* is no more.

```
CREATE TABLE teams (
    id INT,
    league_id INT NOT NULL,
    trigram CHAR(3) NOT NULL,
    name VARCHAR(255),
    location VARCHAR(255),
    PRIMARY KEY (id),
    CONSTRAINT team_unique UNIQUE (league_id, trigram),
    FOREIGN KEY (league_id)
        REFERENCES leagues (id) ON DELETE CASCADE
);

-- @weak
CREATE TABLE team_stats (
    id INT,
    team_id INT NOT NULL,
    year INT NOT NULL,
    team_stat_tactique_id INT NOT NULL,
    stat_id INT NOT NULL,
    pace NUMBER NULL,
    PRIMARY KEY (id),
    CONSTRAINT team_stat_unique UNIQUE (team_id, year, team_stat_tactique_id),
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE,
    FOREIGN KEY (team_stat_tactique_id)
        REFERENCES team_stat_tactiques (id) ON DELETE CASCADE,
    FOREIGN KEY (stat_id)
        REFERENCES stats (id) ON DELETE CASCADE
);
```

2.2 Import the data from the given CSV files into the created database

As mentioned before, we are using *Ruby on Rails* as the application framework in order to build the final application. Below is the very few step required to set it up correctly, as we did it for our local machines. The database configuration is found into the *config/database.yml* file into the *Rails* application (called *nba*). It's set up to be the same credentials for the local and remote (@EPFL) database for simplicity.

2.2.1 Setting up the schema

The following commands will execute the schema creation and loads some initial data (like *leagues*, *conferences*, and so on).

```
$ sqlplus DB2012_G06/DB2012_G06@XE < document/sql/schema.sql
$ sqlplus DB2012_G06/DB2012_G06@XE < document/sql/data.sql
```

2.2.2 Setting up the Rails application

The application dependencies are listed in the *Gemfile* and you should use *bundler* to handle it. It's as easy as the following.

```
$ sudo gem install rails bundler
$ cd nba
$ bundle install
```

2.2.3 Importing the data

The next commands are creating some extra tables (required by the admin interface) and starts parsing the CSV files (from the *dataset* directory).:

```
$ cd nba
$ # locally
$ rake db:migrate
$ rake import:all
$ # remotely
$ RAILS_ENV=production rake db:migrate
$ RAILS_ENV=production rake import:all
```

NB: A faster way of loading data would be to transform the CSV into ready-to-be-inserted CSV and bulk loading them. As we don't trust that much the input data, we went for the much more sluggish approach that performs individual inserts. In the end, it's way easier to debug and it teaches Zen.

2.2.4 Workarounds made to the dataset

We didn't touch the initial file and applied all the workarounds into the import script directly.

Missing teams

Some teams are missing from the CSV, so we are creating them before starting importing everything.

```
missing_teams = {
  "ABA" => %w(NYJ),
  "NBA" => %w(NEW NOR PHW SAN SL1 TAT TOT)
}
missing_teams.each do |league, teams|
  league = League.find_by_name(league)
  teams.each do |trigram|
    puts Team.create(
      :trigram => trigram,
      :league => league
    )
  end
end
```

The Floridians

That team doesn't have any location for historic reasons.

```
# https://en.wikipedia.org/wiki/Miami_Floridians
if row["location"] == "Floridians" then
  row["name"] = row["location"]
  row["location"] = nil
end
```

Non-breaking space

There are some silly characters into the CSV.

```
# One of the coach contains a non-breaking space (nbsp).
row["coachid"] = row[0].tr(" ", " ").strip
```

Magic Johnson and Marques Johnson

Two players are sharing the same *ilkid* and it's a mistake in the data.

```
# Fix a buggy ilkid
if (row["ilkid"] == "JOHNSMA01" and row["firstname"] == "Marques") then
  row["ilkid"] = "JOHNSMA02"
end
```

One entry is a buggy duplicate

In the all star file, there is a duplicate entry that we decided to simply ignore.

```
# Ignore this particular entry
if (row["ilkid"] == "THOMPDA01" and row["year"] == "1982" and row["tpm"] == "NULL") then
  next
end
```

2.3 Accommodate the import of new data in the database they created in the 1st deliverable

To us, this point is about having an admin interface. We've chosen ActiveAdmin which enable easy and nifty automagic interface in respect of the defined model (ActiveRecord).

That's enough for adding simple data. To bulk import more CSV, the import script will just do it by replacing the existing CSV files and running the *import:all* command or any of the per CSV file ones, like: *import:teams*. **NB:** we are assuming it would work but weren't able to test this. You've been warned.

2.3.1 Screenshots

Find below some screenshots of the basic CRUD it enables.

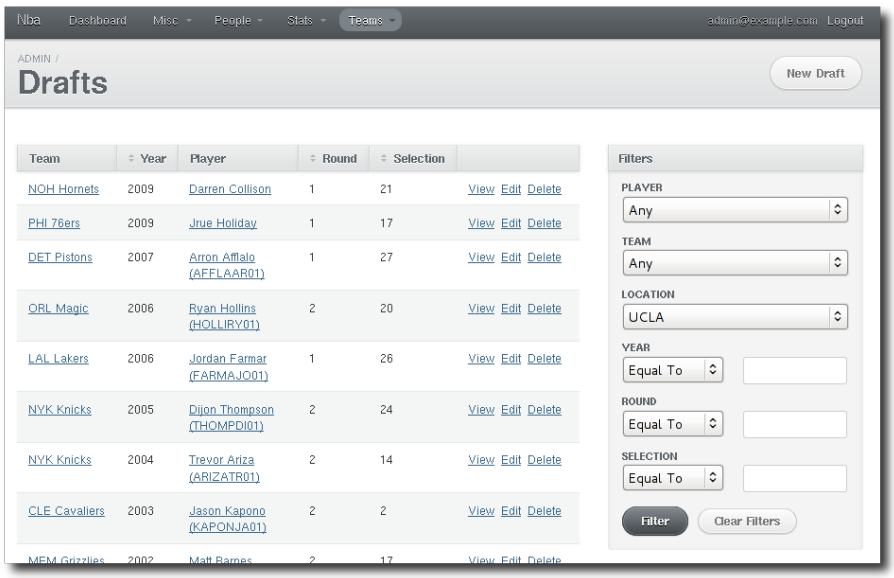


Figure 2.1: Listing all the drafts to NBA made by UCLA.

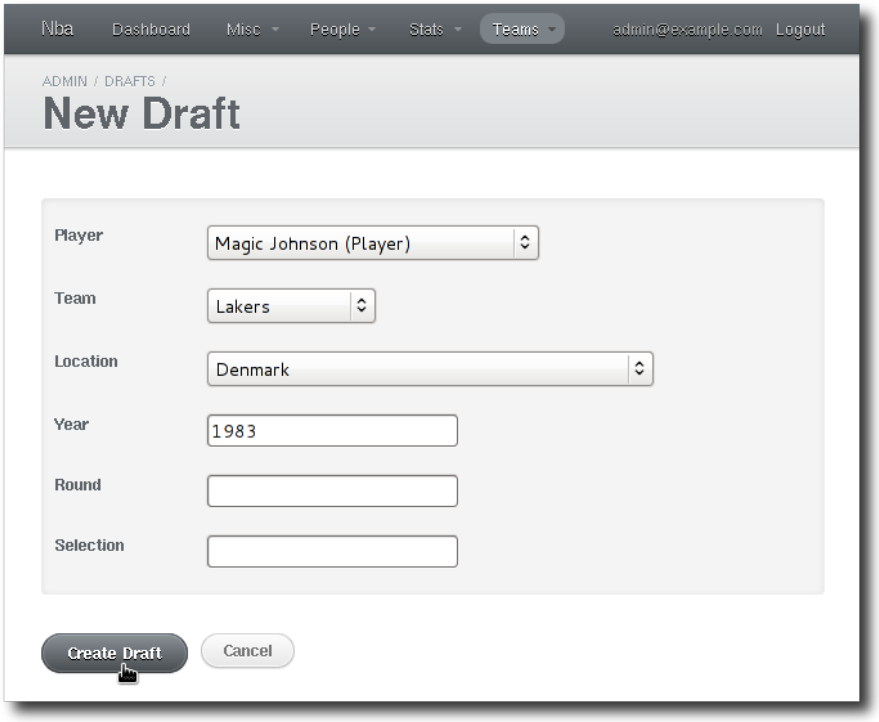


Figure 2.2: Creating a new draft for Magic Johnson.

2.4 Implement the simple search queries

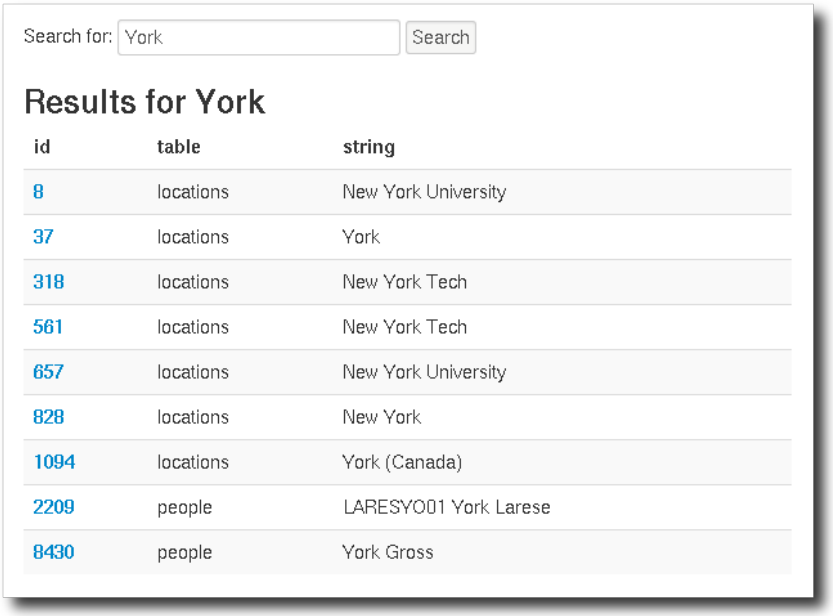
The SQL command is right below. Without any external fulltext search engine, we have to perform a *LIKE '%term%'* on any candidates fields of each tables we would like to be searchable. As it's basically *n* queries, we join them together with the table name to be able to figure out where does it come from.

```
(SELECT
    'teams' tname, id, concat(concat(trigram, ' '), name) str
FROM teams
WHERE
    concat(concat(trigram, ' '), name) LIKE ?

) UNION (

SELECT
    'locations' tname, id,  name str
FROM locations
WHERE name LIKE ?

) UNION (

SELECT
    'people' tname,
    id,
    concat(concat(concat(concat(ilkid, ' '), firstname), ' '), lastname) str
FROM people
WHERE
    concat(concat(concat(concat(ilkid, ' '), firstname), ' '), lastname) LIKE ?

) UNION (

SELECT
    'leagues' tname, id, name str
```

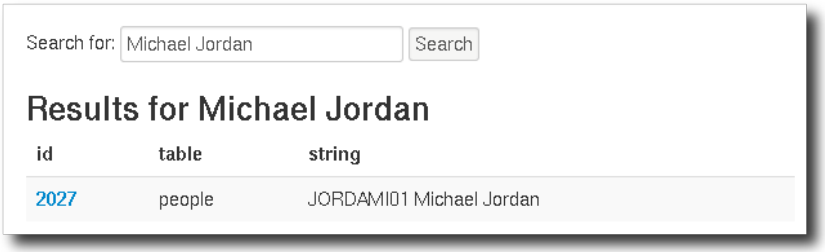
```
FROM leagues
WHERE name LIKE ?

) UNION (

SELECT 'conferences' tname, id, name str
FROM conferences
WHERE name LIKE ?)
```

2.4.1 Screenshots

Find below some screenshot of the search in action. The webpage shows the SQL command ran as well.



Below an example of searching a value made from multiple fields. It's very basic but doing better really requires a better strategy than *LIKE*. There is tons of brilliant softwares that does that very well (Lucene, Solr, Sphinx, Xapian, . . .).



2.4.2 Implement the follow-up search queries of the result of the initial search

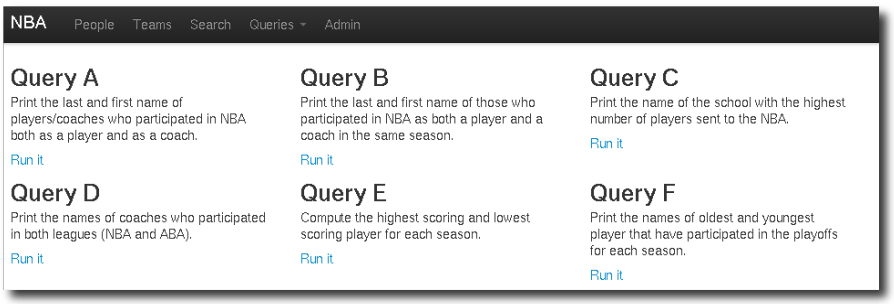
The result of the initial search may look like this.

table name	id	string
people	4050	JAMESMA01 Max Jameson
teams	20	CHI Bulls
. . .	. . .	. . .

From there, we can display something directly and add a link to the proper view for each line.

2.5 Implement using SQL the following queries

A view as been built for each query, which makes them easier to be ran from the web application.



2.5.1 Query A

> *Print the last and first name of players/coaches who participated in NBA both as a player and as a coach.*

The *League* information is tied to a *Team* since each team belongs to a *League*. A *Coach* is linked to a *Team* via the *Coaches Team* table which express a season as coach for the given *Team*. A season played for a *Team* by a *Player* is expressed using the *Player Season* relation. This query fetches all the *Coaches* and all the *Players* from the given *League* and see the ones who match the same *Person*.

```
CREATE OR REPLACE VIEW query_a AS

SELECT DISTINCT
    p.id, p.firstname, p.lastname
FROM
    people p
    JOIN players pl         ON pl.person_id = p.id
    JOIN player_seasons ps  ON ps.player_id = pl.id
    JOIN teams t            ON t.id = ps.team_id
    JOIN leagues l          ON l.id = t.league_id
    JOIN coaches c          ON c.person_id = p.id
    JOIN coach_seasons cs   ON cs.coach_id = c.id
    JOIN teams t2           ON t2.id = cs.team_id
WHERE
    l.name = 'NBA' AND t2.league_id = l.id
ORDER BY
    p.lastname, p.firstname;
```

2.5.2 Query B

> *Print the last and first name of those who participated in NBA as both a player and a coach in the same season.*

This is is very similar to the *Query A* above with one more restriction. The *Coaches Team* and the *Player Season* have to match the same *year* as well.

```
CREATE OR REPLACE VIEW query_b AS

SELECT DISTINCT
    p.id, p.firstname, p.lastname
FROM
    people p
    JOIN players pl         ON pl.person_id = p.id
    JOIN player_seasons ps  ON ps.player_id = pl.id
    JOIN teams t            ON t.id = ps.team_id
```

```
    JOIN leagues l          ON l.id = t.league_id
    JOIN coaches c          ON c.person_id = p.id
    JOIN coach_seasons cs   ON cs.coach_id = c.id
    JOIN teams t2           ON t2.id = cs.team_id
WHERE
    l.name = 'NBA' AND
    l.id = t2.league_id AND
    ps.year = cs.year
ORDER BY
    p.lastname, p.firstname;
```

2.5.3 Query C

Print the name of the school with the highest number of players sent to the NBA.

In order to get the school (or country for foreign players) information, we have to compute among all the drafts which *location* comes first. For people unfamiliar with *Oracle* (like us), you cannot do a simple *GROUP BY locations LIMIT 1* as *LIMIT* doesn't exist in this RDBMS. The alternative is to compute the *RANK()*. The great advantage of RANK is that it may return more than one results in case of equality.

```
CREATE OR REPLACE VIEW query_c AS

SELECT
    id, name, counter
FROM (
    SELECT
        id, name, counter, RANK() OVER (ORDER BY counter DESC) rank
    FROM (
        SELECT
            il.id, il.name, COUNT(il.id) counter
        FROM
            locations il
            JOIN drafts d  ON d.location_id = il.id
            JOIN teams t  ON t.id = d.team_id
            JOIN leagues l ON l.id = t.league_id
        WHERE
            l.name = 'NBA'
        GROUP BY
            il.id, il.name
    )
)
WHERE
    rank = 1;
```

2.5.4 Query D

Print the names of coaches who participated in both leagues (NBA and ABA).

What we are looking for is simply the intersection of coaches who participated in *NBA* with coaches who participated in the *ABA*. In order to facilitate things, we create two views: *nba_coaches* and *aba_coaches*, and use the INTERSECT operator between those views.

The *[an]ba_coaches* views are just simple *JOIN* with a condition on the league name. **NB:** with a different *JOIN* order, it gives no results. With the same order, we have 45 results.

```
CREATE OR REPLACE VIEW nba_coaches AS

SELECT DISTINCT
    p.id, p.lastname, p.firstname
FROM
    teams t
```

```
    JOIN leagues l        ON l.id = t.league_id
    JOIN coach_seasons cs ON cs.team_id = t.id
    JOIN coaches c        ON c.id = cs.coach_id
    JOIN people p         ON p.id = c.person_id
WHERE
    l.name = 'NBA'
ORDER BY
    p.lastname, p.firstname;


CREATE OR REPLACE VIEW aba_coaches AS

SELECT DISTINCT
    p.id, p.lastname, p.firstname
FROM
    teams t
    JOIN leagues l        ON l.id = t.league_id
    JOIN coach_seasons cs ON cs.team_id = t.id
    JOIN coaches c        ON c.id = cs.coach_id
    JOIN people p         ON p.id = c.person_id
WHERE
    l.name = 'ABA'
ORDER BY
    p.lastname, p.firstname;


CREATE OR REPLACE VIEW query_d AS

SELECT * FROM nba_coaches
 INTERSECT
SELECT * FROM aba_coaches;
```

2.5.5 Query E

Compute the highest scoring and lowest scoring player for each season.

By ranking over the number of points a player has gotten, we can easily compute the highest scoring player of all times, by simply ordering by descending rank and querying the one with the first rank.

Since we want the best and worst player for *each* season, we have to use *PARTITION BY year*, which allows us to rank players for each year. Thus, we create two views, one with the best players by year, and one with the worst players. Those views are named *best_players* and *worst_players* respectively.

The next problem is that, for some seasons, there are ex aequos: there might be two or three players that are the best of a season. Similary for worst players, which happens even more often since scoring 0 points is apparently a common occurence. To counter that, we create views based on *best_players* and *worst_players* but with unique years, taking only the last row for every given year.

With all that done, we're just left with combining the *best_players_unique* and *worst_players_unique* views, simply joining them on the year.

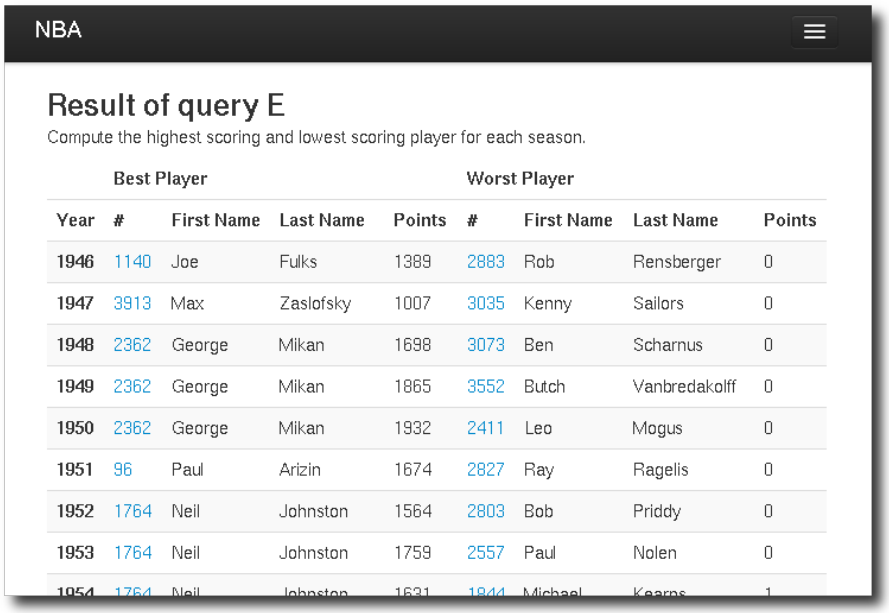
```
CREATE OR REPLACE VIEW best_players AS

SELECT
    player_id best_player_id, player_firstname best_player_firstname,
    player_lastname best_player_lastname, pts best_player_pts, year
FROM (
    SELECT
        player_id, player_firstname, player_lastname, year, pts,
        RANK() OVER (PARTITION BY year ORDER BY pts DESC) r
    FROM (
        SELECT
```

```
            pl.id player_id, p.lastname player_lastname,
            p.firstname player_firstname, psea.year year, stat.pts pts
        FROM
            players pl
            JOIN player_seasons psea ON psea.player_id = pl.id
            JOIN player_stats psta   ON psta.player_season_id = psea.id
            JOIN stats stat          ON psta.stat_id = stat.id
            JOIN people p            ON p.id = pl.person_id
    )
)
WHERE r = 1;


CREATE OR REPLACE VIEW worst_players AS

SELECT
    player_id worst_player_id, player_firstname worst_player_firstname,
    player_lastname worst_player_lastname, pts worst_player_pts, year
FROM (
    SELECT
        player_id, player_firstname, player_lastname, year, pts,
        RANK() OVER (PARTITION BY year ORDER BY pts ASC) r
    FROM (
        SELECT
            pl.id player_id, p.lastname player_lastname,
            p.firstname player_firstname, psea.year year, stat.pts pts
        FROM
            players pl
            JOIN player_seasons psea ON psea.player_id = pl.id
            JOIN player_stats psta   ON psta.player_season_id = psea.id
            JOIN stats stat          ON psta.stat_id = stat.id
            JOIN people p            ON p.id = pl.person_id
    )
)
WHERE r = 1;


CREATE OR REPLACE VIEW best_players_unique AS

SELECT * FROM best_players
WHERE ROWID IN (
    SELECT MAX(ROWID)
    FROM best_players GROUP BY year
);


CREATE OR REPLACE VIEW worst_players_unique AS

SELECT * FROM worst_players
WHERE ROWID IN (
    SELECT MAX(ROWID)
    FROM worst_players
    GROUP BY year
);


CREATE OR REPLACE VIEW query_e AS
SELECT
    bp.year,
    bp.best_player_id, bp.best_player_firstname, bp.best_player_lastname,
    bp.best_player_pts,
    wp.worst_player_id, wp.worst_player_firstname, wp.worst_player_lastname,
    wp.worst_player_pts
```

```
FROM
    best_players_unique bp
    JOIN worst_players_unique wp ON wp.year = bp.year
ORDER BY
    year ASC;
```

Screenshot

Sample output of how it's displayed in the interface.



	Best Player				Worst Player			
Year	#	First Name	Last Name	Points	#	First Name	Last Name	Points
1946	1140	Joe	Fulks	1389	2883	Rob	Rensberger	0
1947	3913	Max	Zaslofsky	1007	3035	Kenny	Sailors	0
1948	2362	George	Mikan	1698	3073	Ben	Scharnus	0
1949	2362	George	Mikan	1865	3552	Butch	Vanbredakolff	0
1950	2362	George	Mikan	1932	2411	Leo	Mogus	0
1951	96	Paul	Arizin	1674	2827	Ray	Ragelis	0
1952	1764	Neil	Johnston	1564	2803	Bob	Priddy	0
1953	1764	Neil	Johnston	1759	2557	Paul	Nolen	0
1954	1764	Neil	Johnston	1631	1844	Michael	Kearns	1

2.5.6 Query F

Print the names of oldest and youngest player that have participated in the playoffs for each season.

This query works exactly like *Query E*, except that instead of *best_players* and *worst_players* we have *youngest_players* and *oldest_players*. The ranking works exactly the same, only ordered by birthdate instead of season points. We also need to weed out duplicates, and to add a test on season types to only get players who participated in the playoffs. Similarly, *youngest_players_unique* and *oldest_players_unique* are joined on year.

```
CREATE OR REPLACE VIEW youngest_players AS

SELECT
    id youngest_id, firstname youngest_firstname, lastname youngest_lastname,
    birthdate youngest_birthdate, year
FROM (
    SELECT
        id, lastname, firstname, year, birthdate,
        RANK() OVER (PARTITION BY year ORDER BY birthdate DESC) r
    FROM (
        SELECT
            pl.id, p.lastname, p.firstname, psea.year, pl.birthdate
        FROM
            players pl
            JOIN player_seasons psea      ON psea.player_id = pl.id
            JOIN player_stats psta        ON psta.player_season_id = psea.id
            JOIN player_season_types psty ON psta.player_season_type_id = psty.id
            JOIN people p ON p.id = pl.person_id
        WHERE
```

```
            psty.name = 'Playoff'
    )
)
WHERE r = 1;


CREATE OR REPLACE VIEW oldest_players AS

SELECT
    id oldest_id, firstname oldest_firstname, lastname oldest_lastname,
    birthdate oldest_birthdate, year
FROM (
    SELECT
        id, lastname, firstname, year, birthdate,
        RANK() OVER (PARTITION BY year ORDER BY birthdate ASC) r
    FROM (
        SELECT
            pl.id, p.lastname, p.firstname, psea.year, pl.birthdate
        FROM
            players pl
            JOIN player_seasons psea      ON psea.player_id = pl.id
            JOIN player_stats psta        ON psta.player_season_id = psea.id
            JOIN player_season_types psty ON psta.player_season_type_id = psty.id
            JOIN people p                 ON p.id = pl.person_id
        WHERE
            psty.name = 'Playoff'
    )
)
WHERE r = 1;


CREATE OR REPLACE VIEW youngest_players_unique AS

SELECT *
FROM youngest_players
WHERE ROWID IN (
    SELECT MAX(ROWID)
    FROM youngest_players
    GROUP BY year
);


CREATE OR REPLACE VIEW oldest_players_unique AS

SELECT *
FROM oldest_players
WHERE ROWID IN (
    SELECT MAX(ROWID)
    FROM oldest_players
    GROUP BY year
);


CREATE OR REPLACE VIEW query_f AS

SELECT
    yp.year,
    yp.youngest_id, yp.youngest_firstname, yp.youngest_lastname,
    yp.youngest_birthdate,
    op.oldest_id, op.oldest_firstname, op.oldest_lastname, op.oldest_birthdate
FROM
    youngest_players yp
    JOIN oldest_players op ON op.year = yp.year
```

```
ORDER BY
    year ASC;
```

2.6 Build an interface to access and visualize the data

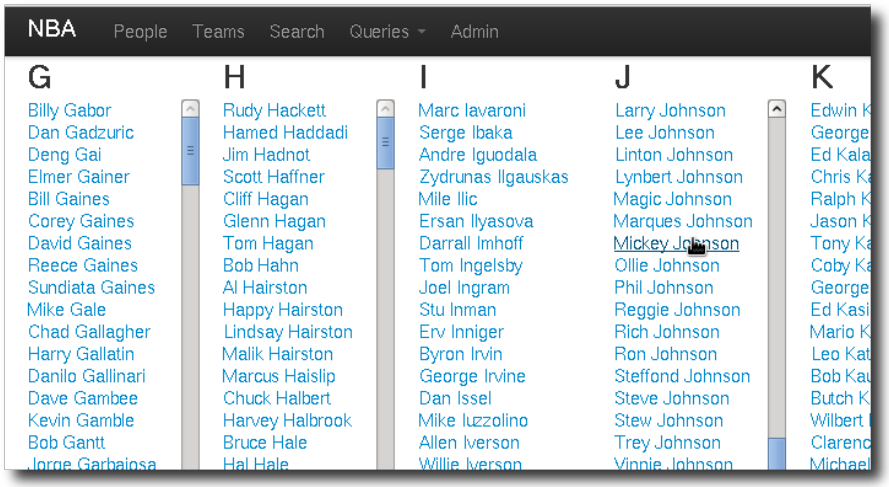
Run the next command to start the webserver.

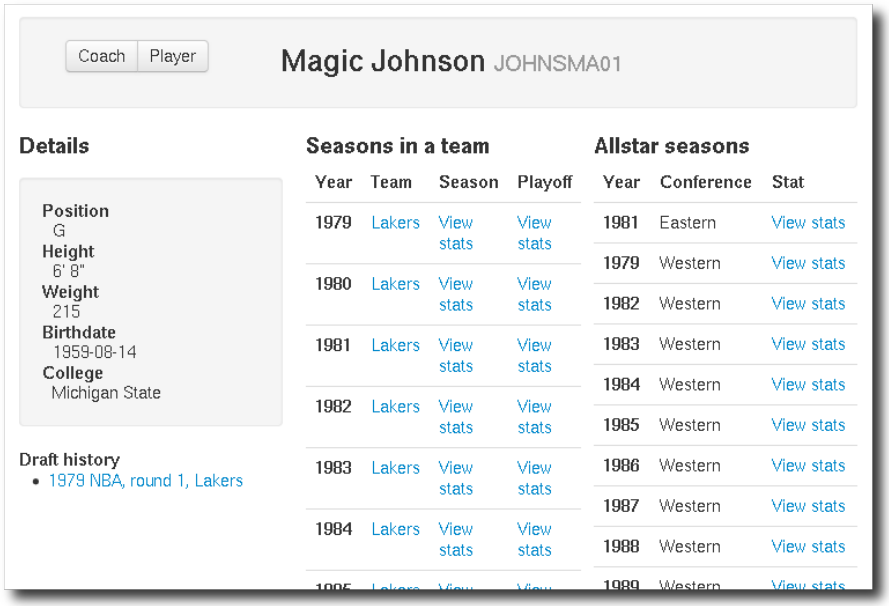
```
$ cd nba
$ # the local database
$ rails server
$ # the remote database at the EPFL
$ rails server -e production
```

2.6.1 Screenshots

Find below some screenshots of the view that exists outside the admin interface. **NB:** they may not reflect the final code.

People





Coach Player

Magic Johnson JOHNSMA01

Details

Position
G
Height
6' 8"
Weight
215
Birthdate
1959-08-14
College
Michigan State

Draft history
- 1979 NBA, round 1, Lakers

Seasons in a team

Year	Team	Season	Playoff
1979	Lakers	View stats	View stats
1980	Lakers	View stats	View stats
1981	Lakers	View stats	View stats
1982	Lakers	View stats	View stats
1983	Lakers	View stats	View stats
1984	Lakers	View stats	View stats

Allstar seasons

Year	Conference	Stat
1981	Eastern	View stats
1979	Western	View stats
1982	Western	View stats
1983	Western	View stats
1984	Western	View stats
1985	Western	View stats
1986	Western	View stats
1987	Western	View stats
1988	Western	View stats
1989	Western	View stats

Coach Player

Magic Johnson JOHNSMA01

Stats

Type	Win	Loss
Overall	5	10
Playoff	0	0

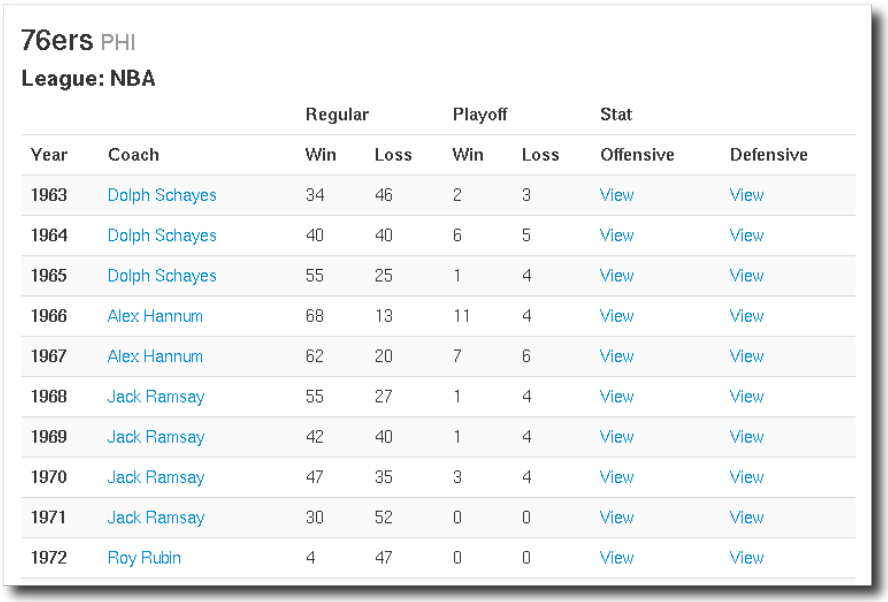
Career

		Regular		Playoff	
Year	Team	Win	Loss	Win	Loss
1993	Lakers	5	10	0	0

Teams

NBA
- Duffey Packers (AND)
- Hawks (ATL)
- Bullets (BA1)
- Bullets (BAL)
- Celtics (BOS)
- Braves (BUF)
- Bullets (CAP)
- Stags (CH1)
- Zephyrs (CH2)
- Packers (CH3)
- Hornets (CHA)
- Bulls (CHI)
- Bobcats (CHR)
- Royals (CIN)
- Rebels (CL1)
- Cavaliers (CLE)
- Mavericks (DAL)
- Falcons (DE1)
- Nuggets (DEN)
- Pistons (DET)
- Nuggets (DN1)
- Zollner Pistons (FTW)

ABA
- Amigos (ANA)
- Cougars (CAR)
- Rockets (DEN)
- Chaparrals (DLC)
- Floridians (FLA)
- Mavericks (HMV)
- Pacers (IND)
- Colonels (KEN)
- Stars (LAS)
- Floridians (MFL)
- Pros (MMP)
- Sounds (MMS)
- Tams (MMT)
- Muskies (MNM)
- Pipers (MNP)
- Americans (NJA)
- Buccaneers (NOB)
- (NYJ)
- Nets (NYN)
- Oaks (OAK)
- Pipers (PTC)
- Pipers (PTP)



76ers PHI

League: NBA

		Regular		Playoff		Stat	
Year	Coach	Win	Loss	Win	Loss	Offensive	Defensive
1963	Dolph Schayes	34	46	2	3	View	View
1964	Dolph Schayes	40	40	6	5	View	View
1965	Dolph Schayes	55	25	1	4	View	View
1966	Alex Hannum	68	13	11	4	View	View
1967	Alex Hannum	62	20	7	6	View	View
1968	Jack Ramsay	55	27	1	4	View	View
1969	Jack Ramsay	42	40	1	4	View	View
1970	Jack Ramsay	47	35	3	4	View	View
1971	Jack Ramsay	30	52	0	0	View	View
1972	Roy Rubin	4	47	0	0	View	View

2.7 Conclusion

We hit the wall pretty hard by getting to know a little better the data while importing it. Guessing out the data really is by reading some CSV is very poor. We guess it's what people call the *implementation gap*. It was way bigger than we usually thought. Knowing that, we can predict that more changes will occur has we progress further.

Working with *Ruby on Rails* and *Oracle XE* is a pretty hard setup to get. But once it works, it's a real bliss (if you have tons of spare RAM). *ActiveAdmin* offers very quickly a way to browse the imported data, spotting mistakes and *ActiveRecord* makes any relationships querying easy.

Oracle is still pretty new to us and the proposed solution are obtained more using the trial and error method than a one shot query that works directly. Stuff like *RANK*, *PARTITION* are uncommon when you have some insight of alternative RDBMS like MySQL, PostgreSQL or SQLite.

CHAPTER

THREE

DELIVERABLE 3

A series of more interesting queries should be implemented with SQL and/or using the preferred application programming language:

- *Explain the necessities of indexes based on the queries and the query plans that you can find from the system;*
- *Report the performance of all queries and explain the distribution of the cost (based again on the plans;*
- *Visualize the results of the queries (in case they are not scalar);*
- *Build an interface to run queries/insert data/delete data giving as parameters the details of the queries.*

3.1 Post-mortem deliverable 2

Good job with the queries. Is there a reason why all queries are expressed as views? Please include the explanation for this decision in the report.

It seemed easier for us to reuse them automagically from the web UI, but we were mistaken and the final code doesn't do that anymore. Both the SQL queries and the web application were simplified following that change.

Can you explain the remark about "different join order not giving any results" in query D? Please include this explanation in the final version of the report. Keep in mind that you're not getting the correct result (should be indeed 0 rows).

Actually, it seemed fishy that a query would give 0 results, and I have to admit I (Amos) am guilty of trial-and-error on this one. Learning Oracle was very interesting and I finally went with the simplest solution I could think of: an intersect.

Keep up the good work! (at least you're gonna read this...)

Indeed!

3.1.1 Changes to the schema

Many simplifications were made mostly for the sake of *keeping it stupid simple* but also because we gained knowledge of the data as we started used them more.

At the end of this section, you'll find the updated schema both picture (UML-like diagram) and SQL commands (without the denormalized stuff that will be describe after that).

The *Coach* is dead, long live the *Coach*

The *Coach* entity was in the end an empty shell pointing to a *Person*. So we dropped it. As you'll see below, it came back as a denormalization entity.

Same fate for the *Player*

After dropping the *Coach* it made sense to also drop the *Player*. Its data got merged into the *Person* entity. Not every *Player* had their `birthdate` or `height` and `weight` information and it could also be useful in the *Coach* use case (besides the `position` attribute).

Again, *Player* came back to contain denormalized data of the player's career stats.

Overkill *Stat*

Everything under the *Stat* umbrella has been moved into their more specific entities: *PlayerStat*, *PlayerAllstart* and *TeamStat*. Even if they have many similarities it doesn't make sense to add that much complexity for so little improvement.

I would blame the *Object-Oriented Programming* kind of design we are practicing for years and corrupted our young minds. It doesn't really apply to this use case.

> *"Object-oriented design is the roman numerals of computing."* — Rob Pike

TeamSeason is the new *TeamStat*

The team statistics for each year were separated between *Defensive* and *Offensive* stats (badly called the *tactique*) creating two rows for a year. It doesn't make much sense and can only lead to further integrity problems if insertion or deletion fail at keeping a couple for a given year.

The new table is now very close to the original CSV file.

PlayerSeason and *PlayerStat* merged into *PlayerSeason*

The *PlayerSeason* was only creating a key linking *Person*, *Team* and a year which might be seen as the season entity. But no data were associated with that. So that *key* moved into *PlayerStat* which got renamed into *PlayerSeason*. It makes the import much more easier, the schema smaller and some of the queries simpler too, removing joins.

If some denormalization were to be made on that kind of element, it might be as useful to create a linked table and keep the table hierarchy as low as possible.

Missing informations

The *League* was missing from the All Stars seasons (*PlayerAllstar*).

Adding the indices

SQL `INDEX` were created on the following columns:

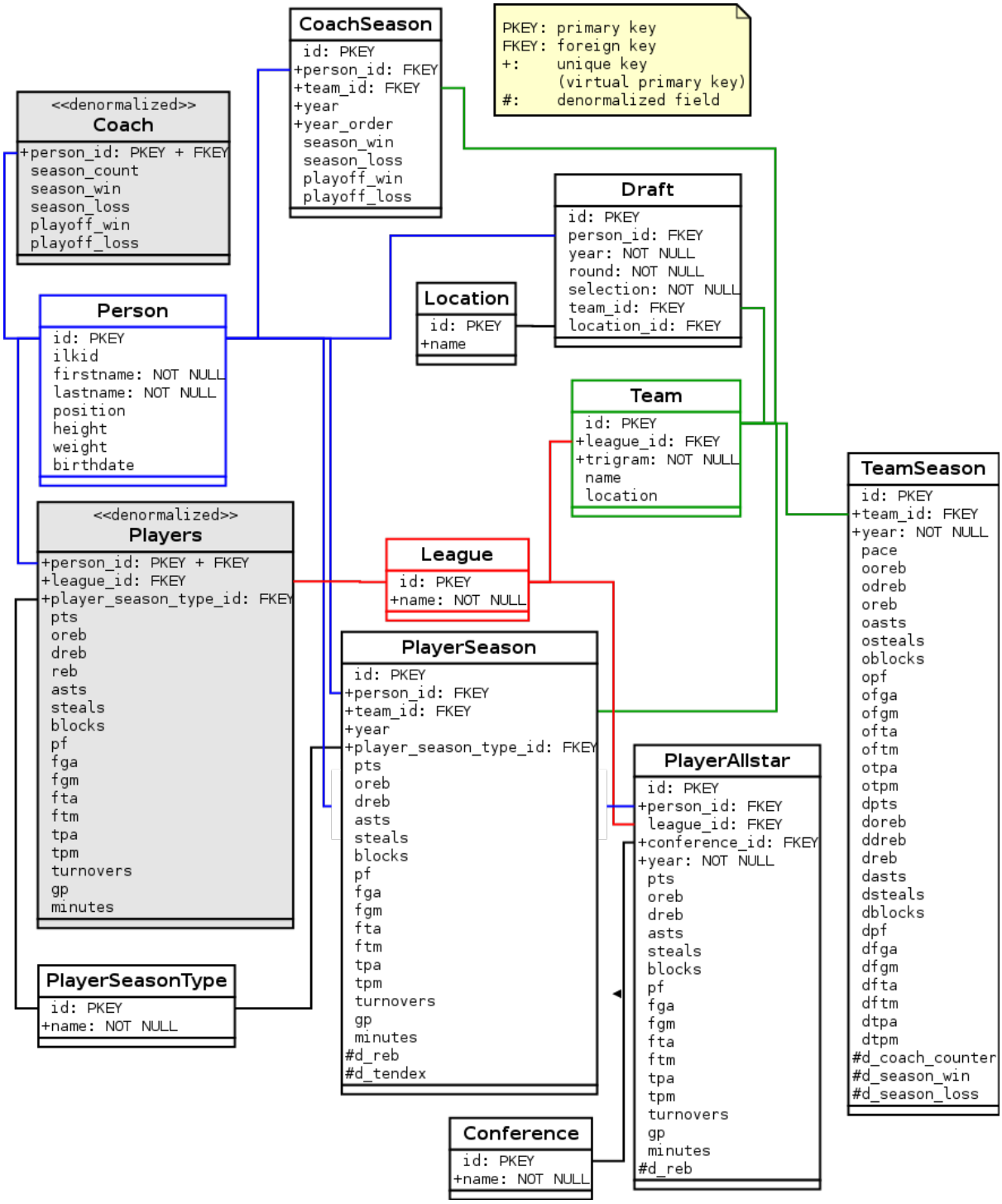
- `year` everywhere it is used since it's acting as a *foreign key* identifying seasons without the burden of maintaining such an entity.
- *Person*'s `ilkid`, `firstname` and `lastname` which is very useful during bulk insertion of new players. The current queries don't take advantages of those.

Were we did not create any indices:

- the small tables with very few items like *Conference*, *League* or *PlayerSeasonType* because:
- the query could be optimized to use the *id* instead of the literal name;
- those tables contain very few elements (here it's 2).

More explainations may be found on the details of the queries below.

Final schema




```
--
-- People
-- ======
--
-- A person can be a player and/or a coach at different time of
-- her life.
--
-- ilkid can be NULL for drafted only players
--

CREATE TABLE people (
```

```
    id INT,
    ilkid VARCHAR(10),
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    position CHAR(1),
    height INT, -- in inches
    weight INT,
    birthdate DATE,
    PRIMARY KEY (id),
    CONSTRAINT person_unique UNIQUE (ilkid, firstname, lastname)
);

CREATE INDEX people_ilkid_idx ON people (ilkid);
CREATE INDEX people_firstname_idx ON people (firstname);
CREATE INDEX people_lastname_idx ON people (lastname);

CREATE SEQUENCE people_seq
    START WITH 1
    INCREMENT BY 1;

--
-- Group of people
-- ===============
--
-- Teams, leagues and stuff
--

-- NBA/ABA
CREATE TABLE leagues (
    id INT,
    name CHAR(3) NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (name)
);

CREATE SEQUENCE leagues_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE conferences (
    id INT,
    name VARCHAR(31) NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (name)
);

CREATE SEQUENCE conferences_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE teams (
    id INT,
    league_id INT NOT NULL,
    trigram CHAR(3) NOT NULL,
    name VARCHAR(255),
    city VARCHAR(255),
    PRIMARY KEY (id),
    CONSTRAINT team_unique UNIQUE (league_id, trigram),
    FOREIGN KEY (league_id)
        REFERENCES leagues (id) ON DELETE CASCADE
);

CREATE SEQUENCE teams_seq
```

```
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE coach_seasons (
    id INT,
    person_id INT NOT NULL,
    team_id INT NOT NULL,
    year INT NOT NULL,
    year_order INT,
    season_win INT,
    season_loss INT,
    playoff_win INT,
    playoff_loss INT,
    PRIMARY KEY (id),
    CONSTRAINT coach_seasons_unique UNIQUE (person_id, team_id, year, year_order),
    FOREIGN KEY (person_id)
        REFERENCES people (id) ON DELETE CASCADE,
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE
);

CREATE INDEX coach_seasons_year_idx ON coach_seasons (year);

CREATE SEQUENCE coach_seasons_seq
    START WITH 1
    INCREMENT BY 1;

--
-- Physical
-- ========
--
-- A school or a country if it's outside the U.S.
--

CREATE TABLE locations (
    id INT,
    name VARCHAR(255),
    PRIMARY KEY (id),
    UNIQUE (name)
);

CREATE SEQUENCE locations_seq
    START WITH 1
    INCREMENT BY 1;

--
-- Drafts
--

CREATE TABLE drafts (
    id INT,
    person_id INT NOT NULL,
    year INT NOT NULL,
    round INT NOT NULL,
    selection INT NOT NULL,
    team_id INT NOT NULL,
    location_id INT NULL,
    PRIMARY KEY (id),
    CONSTRAINT draft_unique UNIQUE (person_id, team_id, location_id, year, round),
    FOREIGN KEY (person_id)
        REFERENCES people (id) ON DELETE CASCADE,
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE,
```

```
    FOREIGN KEY (location_id)
        REFERENCES locations (id) ON DELETE CASCADE
);

CREATE INDEX drafts_year_idx ON drafts (year);

CREATE SEQUENCE drafts_seq
    START WITH 1
    INCREMENT BY 1;


-- Teams seasons
-- =============

CREATE TABLE team_seasons (
    id INT,
    team_id INT NOT NULL,
    year INT NOT NULL,
    pace NUMBER NULL,
    opts INT, -- Offensive
    ooreb INT,
    odreb INT,
    oreb INT,
    oasts INT,
    osteals INT,
    oblocks INT,
    opf INT,
    ofga INT,
    ofgm INT,
    ofta INT,
    oftm INT,
    otpa INT, -- 3pa
    otpm INT, -- 3pm
    dpts INT, -- Defensive
    doreb INT,
    ddreb INT,
    dreb INT,
    dasts INT,
    dsteals INT,
    dblocks INT,
    dpf INT,
    dfga INT,
    dfgm INT,
    dfta INT,
    dftm INT,
    dtpa INT, -- 3pa
    dtpm INT, -- 3pm
    PRIMARY KEY (id),
    CONSTRAINT team_season_unique UNIQUE (team_id, year),
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE
);

CREATE INDEX team_seasons_year_idx ON team_seasons (year);

CREATE SEQUENCE team_seasons_seq
    START WITH 1
    INCREMENT BY 1;

--
-- Players seasons
-- ===============
```

```
CREATE TABLE player_season_types (
    id INT,
    name VARCHAR (31),
    PRIMARY KEY (id),
    UNIQUE (name)
);

CREATE SEQUENCE player_season_types_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE player_seasons (
    id INT,
    person_id INT NOT NULL,
    team_id INT NOT NULL,
    year INT NOT NULL,
    player_season_type_id INT NOT NULL,
    gp INT,
    minutes INT,
    pts INT,
    oreb INT,
    dreb INT,
    asts INT,
    steals INT,
    blocks INT,
    turnovers INT,
    pf INT,
    fga INT,
    fgm INT,
    fta INT,
    ftm INT,
    tpa INT, -- 3pa
    tpm INT, -- 3pm
    PRIMARY KEY (id),
    CONSTRAINT player_season_unique UNIQUE (
        person_id, team_id, year, player_season_type_id
    ),
    FOREIGN KEY (person_id)
        REFERENCES people (id) ON DELETE CASCADE,
    FOREIGN KEY (team_id)
        REFERENCES teams (id) ON DELETE CASCADE,
    FOREIGN KEY (player_season_type_id)
        REFERENCES player_season_types (id) ON DELETE CASCADE
);

CREATE INDEX player_seasons_year_idx ON player_seasons (year);

CREATE SEQUENCE player_seasons_seq
    START WITH 1
    INCREMENT BY 1;

CREATE TABLE player_allstars (
    id INT,
    person_id INT NOT NULL,
    conference_id INT NOT NULL,
    league_id INT NOT NULL,
    year INT NOT NULL,
    gp INT,
    minutes INT,
    pts INT,
    oreb INT,
    dreb INT,
    asts INT,
```

```
    steals INT,
    blocks INT,
    turnovers INT,
    pf INT,
    fga INT,
    fgm INT,
    fta INT,
    ftm INT,
    tpa INT, -- 3pa
    tpm INT, -- 3pm
    PRIMARY KEY(id),
    CONSTRAINT player_allstars_unique UNIQUE (person_id, conference_id, year),
    FOREIGN KEY (person_id)
        REFERENCES people (id) ON DELETE CASCADE,
    FOREIGN KEY (conference_id)
        REFERENCES conferences (id) ON DELETE CASCADE,
    FOREIGN KEY (league_id)
        REFERENCES leagues (id) ON DELETE CASCADE
);

CREATE INDEX player_allstars_year_idx ON player_allstars (year);

CREATE SEQUENCE player_allstars_seq
    START WITH 1
    INCREMENT BY 1;
```

3.1.2 Changes to the queries

Since the schema changed, we had to change the previously done queries. The report for part 2 will still reflect the state at that point of time although the queries might not be exactly the same.

Queries **A** and **B** have trivial changes regarding the deletion of *Coach* and *Player*, one less join has to be done simplying the final query.

Queries **E** and **F** are also doing less `JOIN` since *Player* are no more (its data has being moved into *Person*) and everything that were into *Stat* is now into *PlayerStat*.

Query C

This one changed a lot since the definition of how the *Draft* are counted got clearer. It also uses the powerful `PARTITION` method to cut some overkill sub-`SELECT`. Only the last *Draft* into a specified league *League* is kept and thus counts.

```
SELECT l.id, l.name, lc.counter
FROM (
    SELECT id, counter, RANK() OVER (ORDER BY counter DESC) r
    FROM (
        SELECT location_id id, COUNT(*) counter
        FROM (
            SELECT
                d.id, CONCAT(year, round) yr, location_id,
                MAX(CONCAT(year, round)) OVER (PARTITION BY person_id) last_yr
            FROM drafts d
            JOIN teams t ON t.id = d.team_id
            JOIN leagues l ON l.id = t.league_id
            WHERE l.name = 'NBA'
        )
        WHERE
            yr = last_yr
        GROUP BY
            location_id
```

```
    )
) lc
JOIN locations l ON l.id = lc.id
WHERE r = 1
ORDER BY name ASC;
```

Query D

This query is very straight-forward. Using only two joins per league, we make two different selects, one for all NBA coaches, and one for all ABA coaches. By computing the intersection of those two queries through the `INTERSECT` Oracle SQL statement, we keep only coaches who participated in both leagues. As it turns out, there are none!

Had Oracle SQL not the `INTERSECT` statement we should have had proceeded otherwise: perhaps by selecting all NBA coaches where `coach_id IN aba_coaches`.

```
FROM people
WHERE id IN (
    SELECT DISTINCT person_id
    FROM coach_seasons cs
    JOIN teams t ON t.id = cs.team_id
    JOIN leagues l ON l.id = t.league_id AND l.name = 'NBA'
INTERSECT
    SELECT DISTINCT person_id
    FROM coach_seasons cs
    JOIN teams t ON t.id = cs.team_id
    JOIN leagues l ON l.id = t.league_id AND l.name = 'ABA'
)
ORDER BY lastname, firstname;
```

3.1.3 Importing data

> *As per your comment, the logic in 'import.rake' is quite hardcore. I'm not questioning at all your choice—as a matter of fact, in the same scenario I tend to attack the problem in the very same way (especially because the code plays the role of implicit documentation for each data transformation). Still, for the sake of completeness, I've to mention two other options:*
>
> - *manipulating the .csv with Excel/LibreOffice Calc is a viable and usually quicker solution (but worse in terms of maintainability).*
> - *instead of importing directly into the tables of your final DB schema, you could create a temporary table for each .csv file (same schema, no constraints) and ALTER them progressively. This usually leads to less LOC (being SQL more expressive than Ruby).*

This part was totally redone almost from scratch using `sqlldr`, `LOAD DATA` and `INSERT ALL INTO ... SELECT FROM`. It reduced the importing time from more than one hour to around 2 minutes, a 30x improvement. Time improvement was the main goal here to give us more flexibility in playing with changes in the schema. Manipulating the CSV (more than changing the line ending) wasn't an option as it would make things harder to maintain.

A new method has been added `import:schema` which runs the three SQL files containing the commands about dropping the tables, sequences and procedures (`drop.sql`), creating the tables, sequences and procedures (`schema.sql`) and also importing some initial data (`data.sql`) like leagues, conferences and such.

Loading data with SQLLDR 101

How it works for any CSV file:

- First, a table is created for the CSV file with all fields as `VARCHAR2(255)` as the CSV contains text.

- Then a *control.txt* file is created containing the SQL code to load the data. Check the code below. That code says that the fields are separated using the comma (`,`) and will convert any string `'NULL'` into the proper SQL `NULL` value:

```
    control = '.control.txt'
    c = File.open(control, 'w+')

    # Replacing "NULL" with proper NULL hopefully nobody's called that way
    # but... http://stackoverflow.com/q/4456438/122978
    sqlfields = fields.map do |field|
      "#{field} \"DECODE(:#{field}, 'NULL', NULL, :#{field})\""
    end
    c.write "
LOAD DATA INFILE '#{csv}'
TRUNCATE
INTO TABLE #{table}
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(#{sqlfields.join(', ')})
"
    c.close
```

- Next step is the `sqlldr` call, which is a call to the executable with some arguments like the `control.txt` file, the `userid` being the connection string and `skip` which is set to 1 telling it to ignore the first line containing the column headings.
- Then the `control.txt` file is deleted.
- At this point, data is inserted into the *real* tables using a simple `INSERT INTO` (`#{tmp}` is replaced by the _temporary_ table name):

```
INSERT INTO locations (id, name)
  SELECT locations_seq.NEXTVAL, draft_from
  FROM (
    SELECT DISTINCT TRIM(draft_from) draft_from
    FROM #{tmp}
  )
```

- Finally the initially created table is deleted. `TEMPORARY TABLE`'s don't seem to work with that use case.

We won't clutter this part with more code here and invite you to take a look at `nba/lib/tasks/import.rake` for more details.

3.1.4 Denormalization

In the phase 1 and 2, we managed to get rid of every duplications, trying to keep the core data considering only consistency of the data stored.

We also knew that for some complex tasks, it'd become much more easier to have precomputed fields, tables. Find below which stuff were *denormalized*, how and why.

We tried to use Materialized View but they are unfortunately not available on Oracle XE. So we used only the more conventional `TRIGGER`. Their usage might not be always adequate since its our first time. Any feedbacks will be appreciated.

Coach

As mentioned before the *Coach* entity was an empty shell and not carrying any data. For the need of some queries and to reflect the CSV file `coach_career.csv` the *Coach* entry as been recreated and contains only denormalized data computed from the *CoachSeason* entities.

It could also become a way to know if a *Person* has acted as coach in his career.

```
CREATE TABLE coaches (
    person_id INT NOT NULL,
    season_count INT,
    season_win INT,
    season_loss INT,
    playoff_win INT,
    playoff_loss INT,
    PRIMARY KEY (person_id),
    FOREIGN KEY (person_id)
        REFERENCES people (id) ON DELETE CASCADE
);

CREATE OR REPLACE TRIGGER coaches_data
AFTER INSERT OR UPDATE OR DELETE ON coach_seasons
FOR EACH ROW
DECLARE
  c INT;
  r_id coaches.person_id%type := NULL;
  r_season_count coaches.season_count%type := 0;
  r_season_win coaches.season_win%type := 0;
  r_season_loss coaches.season_loss%type := 0;
  r_playoff_win coaches.playoff_win%type := 0;
  r_playoff_loss coaches.playoff_loss%type := 0;
BEGIN
  IF UPDATING OR INSERTING THEN
    r_id := :new.person_id;
    r_season_count := 1;
    r_season_win := :new.season_win;
    r_season_loss := :new.season_loss;
    r_playoff_win := :new.playoff_win;
    r_playoff_loss := :new.playoff_loss;
  END IF;

  IF UPDATING OR DELETING THEN
    r_id := :old.person_id;
    r_season_count := r_season_count - 1;
    r_season_win := r_season_win - :old.season_win;
    r_season_loss := r_season_loss - :old.season_loss;
    r_playoff_win := r_playoff_win - :old.playoff_win;
    r_playoff_loss := r_playoff_loss - :old.playoff_loss;
  END IF;

  SELECT COUNT(*) INTO c FROM coaches WHERE person_id = r_id;

  IF c = 0 THEN
    INSERT INTO coaches
      (person_id, season_count, season_win, season_loss, playoff_win, playoff_loss)
    VALUES
      (r_id, r_season_count, r_season_win, r_season_loss, r_playoff_win, r_playoff_loss);
  ELSE
    UPDATE coaches SET
      season_count = season_count + r_season_count,
      season_win = season_win + r_season_win,
      season_loss = season_loss + r_season_loss,
      playoff_win = playoff_win + r_playoff_win,
      playoff_loss = playoff_loss + r_playoff_loss
    WHERE
      person_id = r_id;
  END IF;
END coaches_data;
/
```

Player

Again, the *Player* entity was merged into a *Person* to better come back. It's new purpose is to reflect the CSV files `player_career.csv` and `player_playoffs_career.csv` keeping the denormalized sums of all the *PlayerStat* for each type of *PlayerSeason*.

The `TRIGGER`'s are a bit trickier than before mostly because there is much more data involved.

```
CREATE TABLE players (
  person_id INT NOT NULL,
  league_id INT NOT NULL,
  player_season_type_id INT NOT NULL,
  pts INT,
  oreb INT,
  dreb INT,
  reb INT,
  asts INT,
  steals INT,
  blocks INT,
  pf INT,
  fga INT,
  fgm INT,
  fta INT,
  ftm INT,
  tpa INT, -- 3pa
  tpm INT, -- 3pm
  turnovers INT,
  gp INT,
  minutes INT,
  CONSTRAINT players_unique UNIQUE (person_id, league_id, player_season_type_id),
  FOREIGN KEY (person_id)
    REFERENCES people (id) ON DELETE CASCADE,
  FOREIGN KEY (league_id)
    REFERENCES leagues (id) ON DELETE CASCADE,
  FOREIGN KEY (player_season_type_id)
    REFERENCES player_season_types (id) ON DELETE CASCADE
);

CREATE OR REPLACE TRIGGER players_data
AFTER INSERT OR UPDATE OR DELETE ON player_seasons
FOR EACH ROW
DECLARE
  c INT;
  r_person_id players.person_id%type := NULL;
  r_team_id player_seasons.team_id%type := NULL;
  r_league_id players.league_id%type := NULL;
  r_player_season_type_id players.player_season_type_id%type := NULL;
  r_gp players.gp%type := 0;
  r_minutes players.minutes%type := 0;
  r_pts players.pts%type := 0;
  r_oreb players.oreb%type := 0;
  r_dreb players.dreb%type := 0;
  r_reb players.reb%type := 0;
  r_asts players.asts%type := 0;
  r_steals players.steals%type := 0;
  r_blocks players.blocks%type := 0;
  r_turnovers players.turnovers%type := 0;
  r_pf players.pf%type := 0;
  r_fga players.fga%type := 0;
  r_fgm players.fgm%type := 0;
  r_fta players.fta%type := 0;
  r_ftm players.ftm%type := 0;
  r_tpa players.tpa%type := 0;
```

```
  r_tpm players.tpm%type := 0;
BEGIN
  IF INSERTING OR UPDATING THEN
    r_person_id := :new.person_id;
    r_team_id := :new.team_id;
    r_player_season_type_id := :new.player_season_type_id;
    r_gp := :new.gp;
    r_minutes := :new.minutes;
    r_pts := :new.pts;
    r_oreb := :new.oreb;
    r_dreb := :new.dreb;
    r_reb := :new.d_reb;
    r_asts := :new.asts;
    r_steals := :new.steals;
    r_blocks := :new.blocks;
    r_turnovers := :new.turnovers;
    r_pf := :new.pf;
    r_fga := :new.fga;
    r_fgm := :new.fgm;
    r_fta := :new.fta;
    r_ftm := :new.ftm;
    r_tpa := :new.tpa;
    r_tpm := :new.tpm;
  END IF;

  IF UPDATING OR DELETING THEN
    r_person_id := :old.person_id;
    r_team_id := :old.team_id;
    r_player_season_type_id := :new.player_season_type_id;
    r_gp := r_gp - :old.gp;
    r_minutes := r_minutes - :old.minutes;
    r_pts := r_pts - :old.pts;
    r_oreb := r_oreb - :old.oreb;
    r_dreb := r_dreb - :old.dreb;
    r_reb := r_reb - :old.d_reb;
    r_asts := r_asts - :old.asts;
    r_steals := r_steals - :old.steals;
    r_blocks := r_blocks - :old.blocks;
    r_turnovers := r_turnovers - :old.turnovers;
    r_pf := r_pf - :old.pf;
    r_fga := r_fga - :old.fga;
    r_fgm := r_fgm - :old.fgm;
    r_fta := r_fta - :old.fta;
    r_ftm := r_ftm - :old.ftm;
    r_tpa := r_tpa - :old.tpa;
    r_tpm := r_tpm - :old.tpm;
  END IF;

  SELECT league_id INTO r_league_id
  FROM teams
  WHERE id = r_team_id;

  SELECT COUNT(*) INTO c
  FROM players
  WHERE
    person_id = r_person_id AND
    league_id = r_league_id AND
    player_season_type_id = r_player_season_type_id;

  IF c = 0 THEN
    INSERT INTO players (
      person_id, league_id, player_season_type_id, gp, minutes, pts, oreb, dreb,
      reb, asts, steals, blocks, turnovers, pf, fga, fgm, fta, ftm, tpa, tpm
```

```
    ) VALUES (
      r_person_id, r_league_id, r_player_season_type_id, r_gp, r_minutes, r_pts,
      r_oreb, r_dreb, r_reb, r_asts, r_steals, r_blocks, r_turnovers, r_pf,
      r_fga, r_fgm, r_fta, r_ftm, r_tpa, r_tpm
    );
  ELSE
    UPDATE players SET
      gp = gp + r_gp,
      minutes = gp + r_minutes,
      pts = pts + r_pts,
      oreb = dreb + r_oreb,
      dreb = dreb + r_dreb,
      reb = reb + r_reb,
      asts = asts + r_asts,
      steals = steals + r_steals,
      blocks = blocks + r_blocks,
      turnovers = turnovers + r_turnovers,
      pf = pf + r_pf,
      fga = fga + r_fga,
      fgm = fgm + r_fgm,
      fta = fta + r_fta,
      ftm = ftm + r_ftm,
      tpa = tpa + r_tpa,
      tpm = tpm + r_tpm
    WHERE
      person_id = r_person_id AND
      league_id = r_league_id AND
      player_season_type_id = r_player_season_type_id;
  END IF;
END players_data;
/
```

Rebounds and TENDEX

For *PlayerStat* and *PlayerAllstar* (but **not** *TeamSeason*), the `reb` (rebounds), value is the sum of `oreb` and `dreb`, so we were able to remove it which will enforce more integrity. Unfortunately the *TeamSeason* dataset contains data where that condition is not respected because `oreb` and `dreb` are empty.

Since the `TENDEX` value is easily computable for every *PlayerStat* entry a very simple trigger can keep that value up-to-date which will simplify much redundancy among the following queries (and prevent mistakes as well).

PlayerStat will get on extra column called `d_tendex` (`d_` for denormalized) and an attached trigger called upon insertion or update. That value will remain `NULL` if the player never played, which makes sense.

```
ALTER TABLE player_seasons ADD (
    d_reb INT,
    d_tendex NUMBER
);

CREATE OR REPLACE TRIGGER player_seasons_before
BEFORE INSERT OR UPDATE ON player_seasons
FOR EACH ROW
BEGIN
    :new.d_reb := :new.oreb + :new.dreb;
    IF :new.minutes > 0 THEN
        :new.d_tendex := (:new.pts + :new.d_reb + :new.asts + :new.steals +
                          :new.blocks - :new.ftm - :new.fgm - :new.turnovers) /
                          :new.minutes;
    END IF;
END player_seasons_before;
/
```

```
ALTER TABLE player_allstars ADD (
    d_reb INT
);

CREATE OR REPLACE TRIGGER player_allstars_before
BEFORE INSERT OR UPDATE ON player_allstars
FOR EACH ROW
BEGIN
    :new.d_reb := :new.oreb + :new.dreb;
END player_allstars_before;
/
```

TeamSeason and *CoachSeason*

Like the *Coach*, we'd like to keep some information within the *TeamSeason* coming from *CoachSeason*. This information is:

- How many coach seasons do we have;
- How many matches has been won during the regular season;
- How many matches has been lost during the regular season.

No data about the playoffs has been intergrated since *TeamSeason* doesn't reflect any data regarding the playoffs.

This denormalization was initiated by the Query S.

```
ALTER TABLE team_seasons ADD (
    d_coach_counter INT DEFAULT 0,
    d_season_win INT DEFAULT 0,
    d_season_loss INT DEFAULT 0
);

CREATE INDEX team_seasons_d_counter_idx ON team_seasons (d_coach_counter);

CREATE OR REPLACE TRIGGER team_seasons_data
AFTER INSERT OR UPDATE OR DELETE ON coach_seasons
FOR EACH ROW
BEGIN
    IF DELETING OR (UPDATING AND :old.year != :new.year) THEN
        UPDATE team_seasons
        SET
            d_coach_counter = d_coach_counter - 1,
            d_season_win = d_season_win - :old.season_win,
            d_season_loss = d_season_loss - :old.season_loss
        WHERE team_id = :old.team_id AND year = :old.year;
    END IF;

    IF INSERTING OR (UPDATING AND :old.year != :new.year) THEN
        UPDATE team_seasons
        SET
            d_coach_counter = d_coach_counter + 1,
            d_season_win = d_season_win + :new.season_win,
            d_season_loss = d_season_loss + :new.season_loss
        WHERE team_id = :new.team_id AND year = :new.year;
    END IF;
END team_seasons_data;
/
```

3.2 The queries

The `EXPLAIN PLAN` has been computing using *Oracle SQLDeveloper* and the following SQL command:

```
# http://prsync.com/oracle/displaying-the-execution-plan-for-a-sql-statement-26490/
SELECT plan_table_output
FROM table(dbms_xplan.display('plan_table', NULL, 'typical -cost -bytes'))
```

3.2.1 Query G

> *List the name of the schools according to the number of players they sent to the NBA. Sort them in descending order by number of drafted players.*

That query very similar to the query C of deliverable 2 but all the schools (we call them *Location* since it can be a country as well) have to be displayed. It first counts how many drafts a *Location* has and a `LEFT JOIN` is performed on the whole set of *Location*.

Because a *Person* can be drafted in any *League*, Manos told us to keep only the last draft for the given league. It means than one player can be counted twice if he was drafted into the two *Leagues* but not if it was drafted two times in the same *League*.

```
SELECT l1.id, l1.name, l2.counter
FROM locations l1
LEFT JOIN (
    SELECT location_id id, COUNT(*) counter
    FROM (
        SELECT
            d.id, CONCAT(year, round) yr, location_id,
            MAX(CONCAT(year, round)) OVER (PARTITION BY person_id) last_yr
        FROM drafts d
        JOIN teams t ON t.id = d.team_id
        JOIN leagues l ON l.id = t.league_id
        WHERE l.name = 'NBA'
    )
    WHERE
        yr = last_yr
    GROUP BY
        location_id
) l2 ON l1.id = l2.id
ORDER BY
    counter DESC NULLS LAST, name ASC;
```

3.2.2 Query H

> *List the name of the schools according to the number of players they sent to the ABA. Sort them in descending order by number of drafted players.*

Ditto the previous one with ABA instead of NBA.

```
SELECT l1.id, l1.name, l2.counter
FROM locations l1
LEFT JOIN (
    SELECT location_id id, COUNT(*) counter
    FROM (
        SELECT
            d.id, year, round, location_id,
            MAX(CONCAT(year, round)) OVER (PARTITION BY person_id) last
        FROM drafts d
        JOIN teams t ON t.id = d.team_id
        JOIN leagues l ON l.id = t.league_id
```

```
        WHERE l.name = 'ABA'
    )
    WHERE
        CONCAT(year, round) = last
    GROUP BY
        location_id
) l2 ON l1.id = l2.id
ORDER BY
    counter DESC NULLS LAST, name ASC;
```

3.2.3 Query I

> *List the average weight, average height and average age, of teams of coaches with more than* `XXX` *season career wins and more than* `YYY` *win percentage, in each season they coached. (* `XXX` *and* `YYY` *are parameters. Try with combinations:* `{XXX,YYY}={<1000,70%>,<1000,60%>,<1000,50%>,<700,55%>,<700,45%>}` *. Sort the result by year in ascending order.*

The first step here is to compute the season career wins of all coaches (XXX). It is done by simply summing the ratio from season wins to the total number of plays (both wins and defeats). As for the YYY criterion, it is already stored in the `coaches` table as `season_win`.

A first `JOIN` allows us to have XXX and YYY in the same table, along with the coache's identity, its team, and the year of the season in question.

The next step is the trickiest: we have to join the view we just created with the `player_seasons` table, so that we can compute the average of the weight, height, and age of the players who were in the team of a given coach, for a given season. Only the birthdate is stored (as it should), so we have to use the season's year to compute the age of a player at the time of the season.

It turns out that there are NULL weights, so we had to add an additional criterion to prevent those values from corrupting the mean. The query we just discussed is very expensive, because it involves three joins and three `AVG` statements.

However, once that step is done, it's simply a matter of filtering the resulting table according to the :XXX and :YYY parameteres, which are specified from the web interface we built.

As for the front-end, we simply have an HTML form with a select tag, allowing us to pick from the predefined values of :XXX and :YYY that were specified in the project statement. Finally, the view filters out the coach id, first name and last name for duplicates, in order to have a nice display where the seasons of each coach are grouped and easily distinguishable.

```
CREATE OR REPLACE VIEW coach_seasons_percentage AS
    SELECT
        person_id, year, team_id,
        100 * SUM(season_win) / (SUM(season_win) + SUM(season_loss)) win_percentage
    FROM
        coach_seasons cs
    GROUP BY
        person_id, year, team_id;

CREATE OR REPLACE VIEW best_coaches AS
    SELECT
        cp.person_id, cp.win_percentage, c.season_win career_wins, cp.year, cp.team_id
    FROM
        coach_seasons_percentage cp
        JOIN coaches c ON cp.person_id = c.person_id
    ORDER BY
        year;

CREATE OR REPLACE VIEW query_i AS
    SELECT
```

```
        bc.person_id, AVG(p.weight) avg_weight, AVG(p.height) avg_height,
        AVG(ROUND((TO_DATE(bc.year, 'YYYY') - p.birthdate)/365.24,0)) avg_age,
        bc.year, bc.win_percentage, bc.career_wins
    FROM
        best_coaches bc
        JOIN player_seasons ps ON ps.team_id = bc.team_id AND ps.year = bc.year
        JOIN people p ON p.id = ps.person_id AND p.weight IS NOT NULL
        JOIN people p2 ON p2.id = bc.person_id
    GROUP BY
        bc.person_id, bc.year, bc.win_percentage, bc.career_wins;

SELECT
    person_id, firstname, lastname, avg_weight, avg_height, avg_age, year
FROM
    query_i
    JOIN people p ON p.id = person_id
WHERE
    career_wins > :XXX AND win_percentage > :YYY
ORDER BY
    p.lastname, p.firstname, year;
```

3.2.4 Query J

> *List the last and first name of the players which are shorter than the average height of players who have at least 10,000 rebounds and have no more than 12,000 rebounds (if any).*
>
> **Updated description** *we ask you to list the last and first name of the players which have more than 12,000 rebounds and are shorter than the average height of players who have at least 10,000 rebounds (if any).*

First of all, we must compute the total `rebounds` made by a player, here we've take only the one made during *regular* seasons but summed the ABA and NBA scores (for players like Moses Malone `malonmo01` who scored in both).

Then the request happens in two phases:

1. the average `height` is calculated among the players with enough rebounds (`reb`) made during their career (*regular* seasons).
2. are selected the players that are smaller but managed to get more than 12'000 rebounds overall.

Here the denormalized *Player* table is used and helps a lot.

```
CREATE OR REPLACE VIEW player_total_rebounds AS
    SELECT person_id, SUM(reb) total_reb
        FROM
            players pl
            JOIN player_season_types pst ON
                pst.id = pl.player_season_type_id AND
                pst.name = 'Regular'
        GROUP BY person_id;

SELECT p.id, firstname, lastname, height, total_reb
FROM people p
JOIN player_total_rebounds ptr ON ptr.person_id = p.id
WHERE
    height IS NOT NULL AND
    height < (
        SELECT AVG(height)
        FROM
            people p
            JOIN player_total_rebounds ptr ON ptr.person_id = p.id
        WHERE
```

```
            height IS NOT NULL AND
            total_reb >= 10000
    ) AND
    total_reb > 12000
ORDER BY lastname, firstname;
```

3.2.5 Query K

List the last and first name of the players who played for a Chicago team and Houston team.

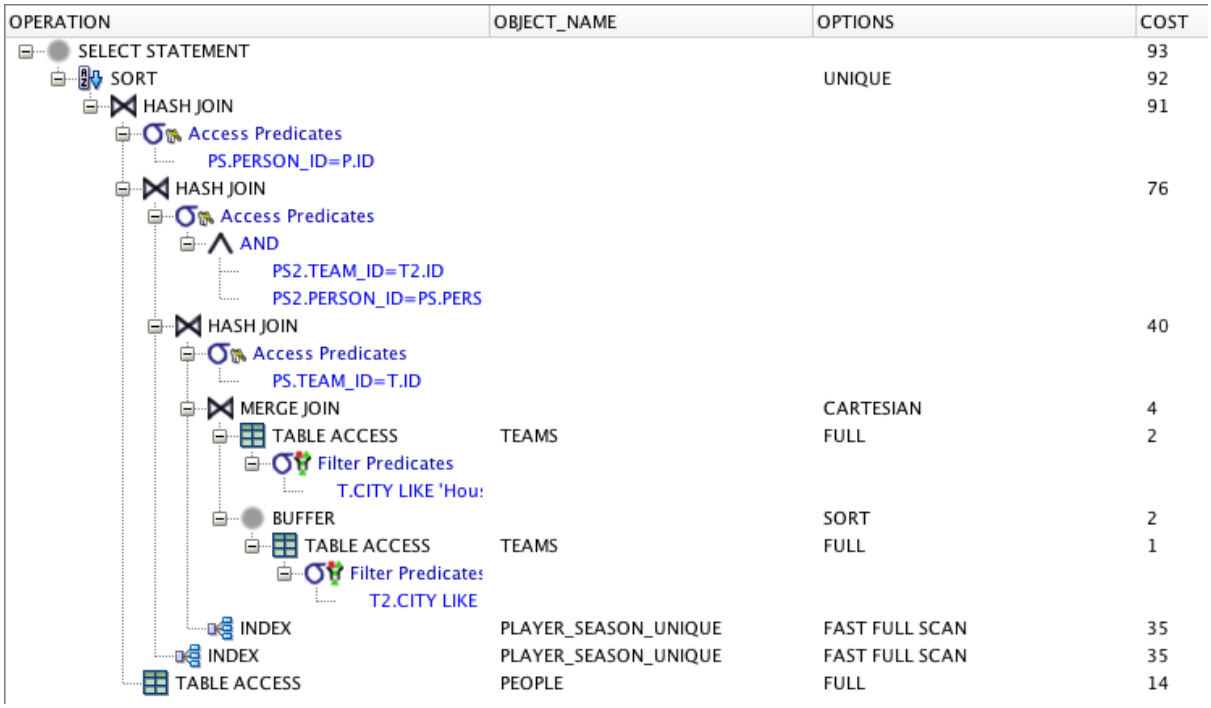
It creates two joins, to filter seasons played in Houston (two teams) or Chicago (four teams). Other strategies are also possible, this one seemed simple enough.

```
SELECT DISTINCT
    p.id, firstname, lastname
FROM
    player_seasons ps
    JOIN teams t ON
        ps.team_id = t.id AND
        t.city LIKE 'Houston'
    JOIN player_seasons ps2 ON ps2.person_id = ps.person_id
    JOIN teams t2 ON
        ps2.team_id = t2.id AND
        t2.city LIKE 'Chicago'
    JOIN people p ON p.id = ps.person_id
ORDER BY
    lastname, firstname;
```

Explain Plan

The Query plan using pure SQL:

Id	Operation	Name	Rows	Time
0	SELECT STATEMENT		178	00:00:01
* 1	HASH JOIN		178	00:00:01
* 2	VIEW		178	00:00:01
* 3	WINDOW SORT PUSHED RANK		178	00:00:01
* 4	TABLE ACCESS FULL	TEAM_SEASONS	178	00:00:01
5	TABLE ACCESS FULL | TEAMS | 107			00:00:01



OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			93
SORT		UNIQUE	92
HASH JOIN			91
Access Predicates			
PS.PERSON_ID=P.ID			
HASH JOIN			76
Access Predicates			
AND			
PS2.TEAM_ID=T2.ID			
PS2.PERSON_ID=PS.PERS			
HASH JOIN			40
Access Predicates			
PS.TEAM_ID=T.ID			
MERGE JOIN		CARTESIAN	4
TABLE ACCESS	TEAMS	FULL	2
Filter Predicates			
T.CITY LIKE 'Hou:			
BUFFER		SORT	2
TABLE ACCESS	TEAMS	FULL	1
Filter Predicates			
T2.CITY LIKE			
INDEX	PLAYER_SEASON_UNIQUE	FAST FULL SCAN	35
INDEX	PLAYER_SEASON_UNIQUE	FAST FULL SCAN	35
TABLE ACCESS	PEOPLE	FULL	14

3.2.6 Query L

List the top 20 career scorers of NBA.

Player is a denormalized table for each *League* and each *PlayerSeasonType*, we just `SUM` all the points (`pts`) made in NBA, `RANK()` and keep the better ones.

```
SELECT
    p.id, firstname, lastname, pts
FROM (
        SELECT
            person_id, SUM(pts) pts,
            RANK() OVER (ORDER BY SUM(pts) DESC) r
        FROM
            players pl
            JOIN leagues l ON
                l.id = pl.league_id AND
                l.name = 'NBA'
        GROUP BY person_id
        ORDER BY pts DESC
    )
    JOIN people p ON p.id = person_id
WHERE
    r <= 20
ORDER BY
    r ASC, lastname ASC, firstname ASC;
```

3.2.7 Query M

For coaches who coached at most 7 seasons but more than 1 season, who are the three more successful? (Success rate is season win percentage: `season_win / (season_win + season_loss)` *. Be sure to count all seasons when computing the percentage.*

Here, we are using the table *coaches* which contains denormalized data built from the *coach_seasons* table and filled via a *TRIGGER*. The `RANK()` method is used and may return more than 3 results in case of a tie.

```
SELECT id, firstname, lastname, rate
FROM (
    SELECT person_id, rate, RANK() OVER (ORDER BY rate DESC) rank
    FROM (
        SELECT person_id, season_win / (season_win + season_loss) rate
        FROM coaches
        WHERE season_count BETWEEN 2 AND 7 -- BETWEEN includes boundaries
    )
)
JOIN people pl ON pl.id = person_id
WHERE rank <= 3
ORDER BY rank ASC;
```

3.2.8 Query N

> *List the last and first names of the top 30 TENDEX players, ordered by descending TENDEX value (Use season stats). (TENDEX=(points+reb+ass+st+blk-missedFT-missedFG-TO)/minutes)*

The *PlayerSeason* already have a precomputed TENDEX but some players did play for many teams during a specific year, thus we must adapte the new computed value. It means re-reading the whole table and that no indices can be used for that.

Then it's a matter of ordering them, ranking them and picking the right number.

```
SELECT
    p.id, firstname, lastname, tendex, r
FROM (
        SELECT
            id, MAX(tendex) tendex,
            RANK() OVER (ORDER BY MAX(tendex) DESC) r
        FROM (
            SELECT person_id id, year, SUM(d_tendex * minutes) / SUM(minutes) tendex
            FROM
                player_seasons ps
                JOIN player_season_types pst ON
                    pst.id = ps.player_season_type_id AND
                    pst.name = 'Regular'
                WHERE d_tendex IS NOT NULL
                GROUP BY person_id, year
                ORDER BY tendex DESC
            )
        GROUP BY id
    ) t
    JOIN people p ON p.id = t.id
WHERE r <= 30
ORDER BY
    r, lastname, firstname ASC;
```

3.2.9 Query O

> *List the last and first names of the top 10 TENDEX players, ordered by descending TENDEX value (Use playoff stats). (TENDEX=(points+reb+ass+st+blk-missedFT-missedFG-TO)/minutes)*

Just like the previous one with less records and a different *PlayerSeasonType*.

```
SELECT
    p.id, firstname, lastname, tendex, r
FROM (
        SELECT
```

```
            id, MAX(tendex) tendex,
            RANK() OVER (ORDER BY MAX(tendex) DESC) r
        FROM (
            SELECT
                person_id id, year,
                SUM(d_tendex * minutes) / SUM(minutes) tendex
            FROM
                player_seasons ps
                JOIN player_season_types pst ON
                    pst.id = ps.player_season_type_id AND
                    pst.name = 'Playoff'
                WHERE d_tendex IS NOT NULL
                GROUP BY person_id, year
                ORDER BY tendex DESC
            )
        GROUP BY id
    ) t
    JOIN people p ON p.id = t.id
WHERE r <= 10
ORDER BY
    r, lastname, firstname ASC;
```

3.2.10 Query P

Compute the least successful draft year – the year when the largest percentage of drafted players never played in any of the leagues.

A quite straightforward query which make usage of `LEFT JOIN` to grab all the *Player* who never played at all by testing the `NULL` content of the joined table. Then a `COUNT(DISTINCT person_id))` to avoid counting players that were drafted many times (various round) and to finish the `RANK()` operation to keep the first one only.

```
SELECT
    year, total
FROM (
    SELECT
        d.year, COUNT(DISTINCT d.person_id) total,
        RANK() OVER (ORDER BY COUNT(d.person_id) DESC) r
    FROM
        drafts d
        LEFT JOIN player_seasons ps ON ps.person_id = d.person_id
    WHERE
        ps.person_id IS NULL
    GROUP BY d.year
)
WHERE r = 1;
```

Explain plan

```
+=====+============================+======================+=======+==========+ |
Id | Operation | Name | Rows | Time | +---+----------------+----------------+-----+--------+ | 0 | SE-
LECT STATEMENT | | 8703 | 00:00:03 | +---+----------------+----------------+-----+--------+ | *1
| VIEW | | 8703 | 00:00:03 | +---+----------------+----------------+-----+--------+ | *2 | WINDOW
SORT PUSHED RANK | | 8703 | 00:00:03 | +---+----------------+----------------+-----+--------+
| 3 | HASH GROUP BY | | 8703 | 00:00:03 | +---+----------------+----------------+-----+--------+ |
4 | VIEW | VW_DAG_0 | 8703 | 00:00:02 | +---+----------------+----------------+-----+--------+ |
5 | HASH GROUP BY | | 8703 | 00:00:02 | +---+----------------+----------------+-----+--------+ |
*6 | HASH JOIN ANTI | | 8703 | 00:00:01 | +---+----------------+----------------+-----+--------+
```

| 7 | TABLE ACCESS FULL | DRAFTS | 8703 | 00:00:01 | +—–+——————————-+——————–—–+———+———–+ | 8 | INDEX FAST FULL SCAN| PLAYER_SEASON_UNIQUE | 26280 | 00:00:01 | +—–+——————————-+——————–—–+———+———–+

```
1. filter("R"=1)
2. filter(RANK() OVER ( ORDER BY NVL(SUM("ITEM_3"),0) DESC )<=1)
6. access("PS"."PERSON_ID"="D"."PERSON_ID")
```

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			85
VIEW			85
Prédicats de filtre			
R=1			
WINDOW		SORT PUSHED RANK	85
Prédicats de filtre			
RANK() OVER (ORDER BY COUNT(*) DESC)			
HASH		GROUP BY	85
HASH JOIN		ANTI	82
Prédicats d'accès			
PS.PERSON_ID=D.PERSON_ID			
TABLE ACCESS	DRAFTS	FULL	13
INDEX	PLAYER_SEASON_UNIQUE	FAST FULL SCAN	68

This query seems to be an heavy one were the two involved tables must be read. In fact, *Oracle* performs that `LEFT JOIN ... WHERE IS NULL` in an optimized way using the *Hash join anti* which in our understanding is a reversed *Hash join* (an optimized join with a bitmap representation). So the `UNIQUE CONSTRAINTS` index is used, for the *PlayerSeason*. That *key* is composed of 4 integers where the *person_id* is the first part of it which means we can use it here.

It's still an heavy request, we decided to drop the information `first_season` and `last_season` from the original dataset and did not recreate it. It might be used here and improve the overall result. It's a pure assumption and must be tested in order to prove it.

3.2.11 Query Q

> *Compute the best teams according to statistics: for each season and for each team compute* `TENDEX` *values for its best 5 players. Sum these values for each team to compute* `TEAM TENDEX` *value. For each season list the team with the best win/loss percentage and the team with the highest* `TEAM TENDEX` *value.*

This *view* is clearly a join of two other *views*:

- One listing for a given year the best team according to the `TEAM TENDEX` value decribed as above;
- the second, for a given year the best team according to the `season_win / season_loss` ratio.

Then, it's only a matter of joining them by year and retrieving all the team information that are usefull to display (`name` or `trigram` if empty).

Those requests are making a great use of `PARTITION` which is used to *cut* by each year picking what's required (the `MAX` value here). But also to select the top *n* of a kind (like the top 5 tendices).

```
CREATE OR REPLACE VIEW team_season_tendices AS
    SELECT id, year, team_tendex
    FROM (
        SELECT
            id, year, team_tendex,
            MAX(team_tendex) OVER (PARTITION BY year) max_team_tendex
        FROM (
```

```
            SELECT
                team_id id, year, SUM(tendex) team_tendex
            FROM (
                SELECT
                    team_id, year, d_tendex tendex,
                    ROW_NUMBER() OVER (PARTITION BY team_id, year ORDER BY
                    d_tendex DESC) r
                FROM
                    player_seasons ps
                    JOIN player_season_types pst ON
                        pst.id = ps.player_season_type_id
                WHERE
                    pst.name = 'Regular' AND
                    d_tendex IS NOT NULL
            )
            WHERE r <= 5
            GROUP BY team_id, year
        )
    )
    WHERE team_tendex = max_team_tendex;

CREATE OR REPLACE VIEW team_season_winlosses AS
    SELECT id, year, winloss
    FROM (
        SELECT
            id, year, winloss,
            MAX(winloss) OVER (PARTITION BY year) max_winloss
        FROM (
            SELECT
                team_id id, year,
                SUM(season_win) / (SUM(season_win) + SUM(season_loss)) winloss
            FROM
                coach_seasons cs
            GROUP BY year, team_id
        )
    )
    WHERE winloss = max_winloss;

SELECT
    tst.year,
    tst.id tid, t1.name tname, t1.trigram ttrigram, team_tendex,
    tswl.id wlid, t2.name wlname, t2.trigram wltrigram, winloss
FROM
    team_season_tendices tst
    JOIN teams t1 ON t1.id = tst.id
    JOIN team_season_winlosses tswl ON tswl.year = tst.year
    JOIN teams t2 ON t2.id = tswl.id
ORDER BY
    year ASC;
```

3.2.12 Query R

> *List the best 10 schools for each of the following categories: scorers, rebounders, blockers. Each school's category ranking is computed as the average of the statistical value for 5 best players that went to that school. Use player's career average for inputs.*

None of the current denormalized table we have hold any `AVG` information so it must be computed in a view that does it for the three values.

Another view will pick an arbitrary *Location* for the *Draft* (we did it in a late rush) and then three dedicated views will compute the average of top 5 (averages) that can be used later on by the final view which picks the top 10 and actually retrieve the *Location* name.

The `:TYPE` is replaced by the actual `VIEW` at run time.

```
CREATE OR REPLACE VIEW player_averages AS
    SELECT
        person_id, AVG(d_reb) rebs, AVG(pts) pts, AVG(blocks) blocks
    FROM
        player_seasons ps
        JOIN player_season_types pst ON
            pst.id = ps.player_season_type_id AND
            pst.name = 'Regular'
    GROUP BY person_id;

CREATE OR REPLACE VIEW player_locations AS
    SELECT
        person_id, MAX(location_id) location_id -- arbitrary choice
    FROM
        drafts d
    GROUP BY person_id;

CREATE OR REPLACE VIEW location_rebs AS
    SELECT
        location_id, AVG(rebs) value,
        RANK() OVER (ORDER BY AVG(rebs) DESC) r
    FROM (
        SELECT
            location_id, rebs,
            ROW_NUMBER() OVER (PARTITION BY location_id ORDER BY rebs DESC) r
        FROM
            player_averages pa
            JOIN player_locations pl ON pl.person_id = pa.person_id
    )
    WHERE r <= 5
    GROUP BY location_id;

CREATE OR REPLACE VIEW location_pts AS
    SELECT
        location_id, AVG(pts) value,
        RANK() OVER (ORDER BY AVG(pts) DESC) r
    FROM (
        SELECT
            location_id, pts,
            ROW_NUMBER() OVER (PARTITION BY location_id ORDER BY pts DESC) r
        FROM
            player_averages pa
            JOIN player_locations pl ON pl.person_id = pa.person_id
    )
    WHERE r <= 5
    GROUP BY location_id;

CREATE OR REPLACE VIEW location_blocks AS
    SELECT
        location_id, AVG(blocks) value,
        RANK() OVER (ORDER BY AVG(blocks) DESC) r
    FROM (
        SELECT
            location_id, blocks,
            ROW_NUMBER() OVER (PARTITION BY location_id ORDER BY blocks DESC) r
        FROM
            player_averages pa
            JOIN player_locations pl ON pl.person_id = pa.person_id
    )
    WHERE r <= 5
    GROUP BY location_id;
```

```
SELECT
    location_id, name, value
FROM
    location_:TYPE
    JOIN locations l ON l.id = location_id
WHERE r <= 10
ORDER BY r ASC, name ASC;
```

3.2.13 Query S

> *Compute which was the team with most wins in regular season during which it changed 2, 3 and 4 coaches.*

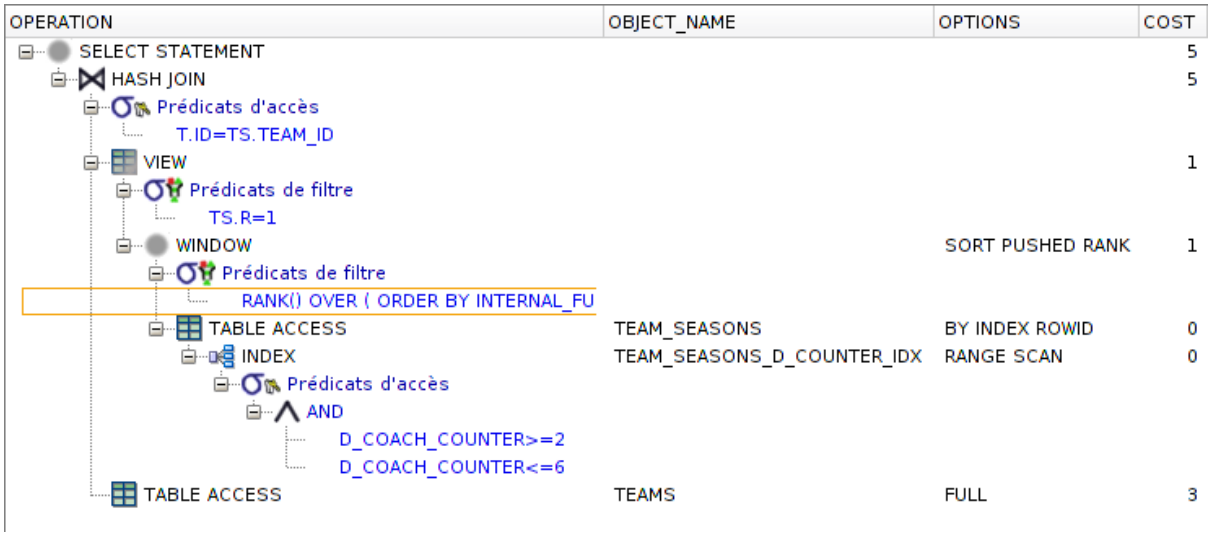
That query is using two denormalized fields added on the *TeamSeason*. Otherwise we would have to read the whole table of *CoachSeason*.

```
SELECT
    team_id, name, trigram, num_coaches, total_wins, year
FROM (
        SELECT
            team_id, year, d_coach_counter num_coaches, d_season_win total_wins,
            RANK() OVER (ORDER BY d_season_win DESC) r
        FROM team_seasons
        WHERE d_coach_counter BETWEEN 2 AND 4
    ) ts
    JOIN teams t ON t.id = ts.team_id
WHERE
    r = 1;
```

Explain Plan

Id	Operation	Name	Rows	Time
0	SELECT STATEMENT		178	00:00:01
*1	HASH JOIN		178	00:00:01
*2	VIEW		178	00:00:01
*3	WINDOW SORT PUSHED RANK		178	00:00:01
4	TABLE ACCESS BY INDEX ROWID	TEAM_SEASONS	178	00:00:01
*5	INDEX RANGE SCAN	TEAM_SEASONS_D_COUNTER_IDX	6	00:00:01
6	TABLE ACCESS FULL	TEAMS	107	00:00:01

```
1. access("T"."ID"="TS"."TEAM_ID")
2. filter("TS"."R"=1)
3. filter(RANK() OVER ( ORDER BY INTERNAL_FUNCTION("D_SEASON_WIN") DESC)<=1)
5. access("D_COACH_COUNTER">=2 AND "D_COACH_COUNTER"<=4)
```



OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			5
HASH JOIN			5
Prédicats d'accès			
T.ID=TS.TEAM_ID			
VIEW			1
Prédicats de filtre			
TS.R=1			
WINDOW		SORT PUSHED RANK	1
Prédicats de filtre			
RANK() OVER (ORDER BY INTERNAL_FU			
TABLE ACCESS	TEAM_SEASONS	BY INDEX ROWID	0
INDEX	TEAM_SEASONS_D_COUNTER_IDX	RANGE SCAN	0
Prédicats d'accès			
AND			
D_COACH_COUNTER>=2			
D_COACH_COUNTER<=6			
TABLE ACCESS	TEAMS	FULL	3

This query seems to be interpreted in the following chronological order:

1. the `RANGE SCAN` using the index on `d_coach_counter`.
2. returning the *TeamSeason* by `ROWID` according to the result of the scan. In total, there are 1337 rows in that table. Only 13% is read (even though many page may have be read since the index is unclustered)
3. the `RANK()` operation is performed
4. a `HASH JOIN` is made between the result of the previous operation (`VIEW`) and the *Team*.
5. Finally the `SELECT` is made.

According the following web article: Interpreting EXPLAIN PLAN tells us that the *Hash join* is way of joining more efficient than *Sort-Merge join* and *Nested Loops* which is a good sign here.

The naive solution would have performed a full table scan on the table *CoachSeason* which is around 1450 rows long and operations like `SUM` and `GROUP BY`. It seems the full access on the *Team* cannot be prevented even though we are matching a *foreign key*.

3.2.14 Query T

> *List all players which never played for the team that drafted them.*

Here a basic usage of the `LEFT JOIN`. It's matching the *Draft* information with the *PlayerSeason* information regarding *Person* and *Team* (via `person_id` and `team_id`). If no matches are made then no *PlayerSeason* are found and thus any of its expected fields are empty (`IS NULL`).

```
SELECT
    p.id, firstname, lastname
FROM
    drafts d
    LEFT JOIN player_seasons ps ON
        ps.team_id = d.team_id
    JOIN people p ON p.id = d.person_id
WHERE
    ps.team_id IS NULL
ORDER BY
    lastname, firstname;
```

3.3 Conclusion

So, it's bit in a hurry that we are writing this little conclusion here. Sorry for that, team work is hard in the industry and very hard in the school context. Do-ocracy doesn't always works.

To conclude, we are happy that we forced us to go with Oracle, like many things it might be a real pain to set up, but then you get access to very powerful tools that can do great stuff. Particulary the OLAP-related onces like *RANK()*, *PARTITION()*, that we hope made a good use. Ruby on Rails and ActiveAdmin helped us focusing on the real stuff, the data and forgetting about the UI which is just another one (for people like us who are used to it).

The one thing I'd like to say about this it only works by doing one iteration after the other, especially when you're evoluting in an relatively unknown context (in regards of the dataset, Oracle itself, ...). *Iterate, iterate, iterate,* ..., no schemas are set in stone and nobody should be scared by starting over. Like we did for the import, with no regrets whatsoever.

In anyways, we learned some stuff about Oracle, why we may just into PostgreSQL more easily in the future. We learned about each other too, why we may to things differently.

Many thanks for supporting our moods, sense of humour, ...