**MET CS 669**

**Fall 1, 2020**

**Term Project Iteration 5**

**October 15, 2020**

**By Iryna Chervachidze**

1. **TOWN YOUTH SOCCER (TYS) Project Iteration 5**

This iteration finalizes all steps of design. Additionally, it populates all necessary tables in the database, creates a history table, a trigger that records values in the history tables when there is a change in the annual fee and solves three business related questions using SQL queries.

* 1. ***The Sign Up Use Case:***

To be able to use the app, each manager and coach need to create an account that would keep his/her user name and password. Depending on the role of the user, each account has different rights with respect to update/delete information as well as keeping track of players’ attendance and skills. In iteration 3 I introduced a Specialization/Generalization relationship with an abstract Person entity. Person has two subtypes, Player and Parent. Both Manager and Coach inherit from the Parent Entity.

Managers collect some personal data about players and collect payment for annual fee and uniforms. Since players are predominantly minors, the managers also collect information on a parent/guardian associated with each player. The payment information is also stored in the database. Coaches take attendance each practice and evaluate players’ skills.

Parents also create their own account to be able to view the feedback from the coaches.

Based on these use cases, I was able to define the following set of structural rules:

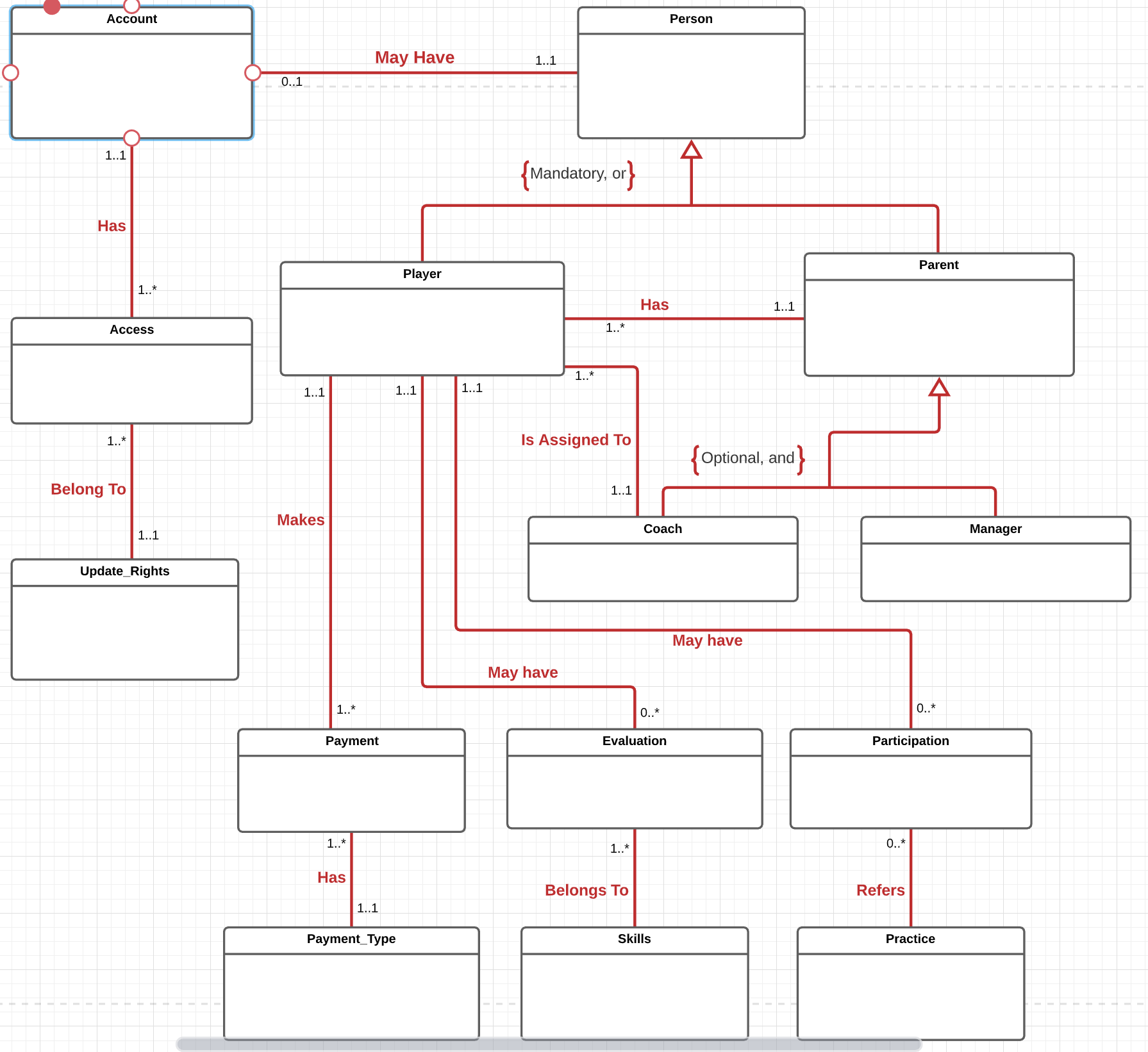
***1.3 TYS Structural Rules***

1. Each Player has one Parent. Each Parent has at least one or more Players.
2. Each Player is assigned to one Coach. Each Coach has one or more Players assigned to him.
3. Each Player has multiple Skills. Each Skill is evaluated for multiple Players. This is a many-to-many relationship and needs an additional bridging entity in its physical model. The bridging entity is Evaluation. Creation of the bridging entity changes the rule slightly. Now, the Player may have many evaluations (in case the player never attends any of the practices, he may have no evaluations at all). However, each evaluation must be associated with one player. Any Skill may belong to a number of evaluations. Each evaluation contains only one skill.
4. Each Player has multiple Practices. Each Practice has multiple Players. This is another many-to-many relationship that requires a bridging entity. The bridging entity that I use here is Participation. Each player may have many instances of Participation. Each participation is associated with just one Player. Each instance of Participation refers to only one Practice. Each Practice may have one or more Participations.
5. Each Player makes multiple Payments. Each Payment is made by one Player.
6. Each Payment has one Payment Type. One Payment Type may be associated with multiple Payments.
7. Each Person must be either a Player or a Parent. Both Player and Parent inherit from Person.
8. Parent can also be a Manager or a Coach, but does not have to be one. Managers and Coaches inherit from a Parent.

All these business rules are summarized in the conceptual model ERD below:

1. **Initial Conceptual Model ERD:**

**Figure 1.Initial Conceptual Model ERD**



1. **Specifying Attributes for Physical Model ERD**

|  |  |  |  |
| --- | --- | --- | --- |
| **Table** | **Attribute** | **Datatype** | **Reasoning** |
| Person | Person\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Person | First\_name | VARCHAR(255) | This is the first name of the person, 255 characters should be enough to store a name |
| Person | Last\_name | VARCHAR(255) | This is the last name of the person, 255 characters should be enough to store a last name |
| Person | Street | VARCHAR(255) | This is the street name of the person’s address, 255 characters should be sufficient to store it |
| Person | City | VARCHAR(255) | This is the city in the person’s address, 255 characters should be sufficient to store it |
| Person | State | VARCHAR(2) | Since each state only requires two letters (such as MA), VARCHAR type with 2 characters is what is needed. |
| Person | Zip | VARCHAR(10) | This is the zip in the person’s address. Even though zip is a number, I will store it as a string of characters because I do not need to perform math operations on the zip. Full zip is only 10 characters long, that’s why I use VARCHAR(10). This will also accommodate for 5 digit zip codes. |
| Person | Is\_player | DECIMAL(1) | The Person can only be either a Player or a Parent. This flag helps the user understand if the person is a player of a parent. It can only have a value 0 or 1, that’s why the type is DECIMAL(1). |
| Person | Is\_parent | DECIMAL(1) | Same as above. The Person can only be either a Player or a Parent. This flag helps the user understand if the person is a player of a parent. It can only have a value 0 or 1, that’s why the type is DECIMAL(1). |
| Player | Person\_id | DECIMAL(12) | A Primary and Foreign Key from the Person table |
| Player | Grade | VARCHAR(2) | Grades can go from K to 12, so using VARCHAR here is appropriate, 2 characters is enough. |
| Player | Parent\_id | DECIMAL(12) | This is a Foreign Key from Parent table |
| Player | Emergency\_first | VARCHAR(255) | Emergency contact first name, just like for the first name of the person VARCHAR(255) should by sufficient. |
| Player | Emergency\_ last | VARCHAR(255) | Emergency contact last name, just like for the last name of the person VARCHAR(255) should by sufficient. |
| Player | Emergency\_ phone | VARCHAR(12) | Emergency contact phone number, since I do not expect any mathematical operations to be performed with it, I assign it to be of type VARCHAR and 12 characters should be sufficient (10 characters for the digits and two characters for the dashes). |
| Player | Coach\_id | DECIMAL(12) | This is a Foreign key to Coach table |
| Player | Travel\_team | DECIMAL(1) | This is a flag that helps understand if the player is on the travel team or not. The only acceptable values are 1 (on the travel team) and 0 (not on the travel team). Therefore DECIMAL(1) is the appropriate type. |
| Parent | Person\_id | DECIMAL(12) | A Primary and Foreign Key from the Person table |
| Parent | Phone\_number | VARCHAR(12) | Parent phone number, since I do not expect any mathematical operations to be performed with it, I assign it to be of type VARCHAR and 12 characters should by sufficient (10 characters for the digits and two characters for the dashes). |
| Parent | email | VARCHAR(32) | Email address of the parent for communication. It a string of characters, therefore VARCHAR is appropriate, up to 32 characters. |
| Parent | Is\_coach | DECIMAL(1) | This is a flag that helps understand if the player is a coach. The only acceptable values are 1 (the parent is a coach) and 0 (the parent is not a coach). Therefore DECIMAL(1) is the appropriate type. |
| Parent | Is\_manager | DECIMAL(1) | This is a flag that helps understand if the player is a manager. The only acceptable values are 1 (the parent is a coach) and 0 (the parent is not a coach). Therefore DECIMAL(1) is the appropriate type. |
| Coach | Person\_id | DECIMAL(12) | A Primary and Foreign Key from the Person table |
| Coach | team | VARCHAR(32) | The name of the team the coach is training. Like any name, it’s a string, so the type is VARCHAR, up to 32 characters. |
| Coach | field | VARCHAR(32) | The name of the field the coach is assigned to. Like any name, it’s a string, so the type is VARCHAR, up to 32 characters. |
| Manager | Person\_id | DECIMAL(12) | A Primary and Foreign Key from the Person table |
| Account | Account\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Account | Person\_id | DECIMAL(12) | Each account belongs to a person. This attribute links account table to Person table. A Foreign Key from the Person table |
| Account | User\_name | VARCHAR(32) | Unique username associated with each account. I allow usernames to be 32 characters long. |
| Account | Password | VARCHAR(255) | Every account has a password. It will be stored in encrypted text format in the database. 255 characters should be a safe limit to store encrypted text. |
| Access | Access\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Access | Account\_id | DECIMALS(12) | Each access belongs to one account. This is a Foreign Key linking to Account table. |
| Access | Rights\_id | DECIMAL(12) | Each access allows the use of one right in the Update\_rights table. This is a Foreign Key linking to Update\_Rights table. |
| Update\_Rights | Rights\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Update\_Rights | Description | VARCHAR(64) | Provides description of rights that are available, such as delete, update, read-only, etc. It is assigned to VARCHAR type up to 64 characters. |
| Payment | Payment\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Payment | Player\_id | DECIMAL(12) | Each payment is made by a player, so this Foreign Key links to the Player table. |
| Payment | Payment\_date | Date | Date on which the payment was made |
| Payment | Pmt\_type\_id | DECIMAL(12) | Each payment is of certain type, such as annual fee payment, or uniform payment. This Foreign Key links to the Table Payment\_Type |
| Payment | Payment\_amount | DECIMAL(6, 2) | This is the amount paid. Evidently, it need to be a DECIMAL type. I allow up to 4 digits with 2 after the decimal point). Club’s annual fee is not big, so I anticipate that 4 digits before the decimal point and 2 after should suffice. |
| Payment\_Type | Pmt\_type\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Payment\_Type | Pmt\_description | VARCHAR(32) | Description of the payment type, such as annual fee, or uniform payment. |
| Evaluation | Evaluation\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Evaluation | Player\_id | DECIMAL(12) | Each evaluation is made for a particular player, so this Foreign Key links to the Player table. |
| Evaluation | Skill\_id | DECIMAL(12) | Each evaluation is made for a particular skill, so this Foreign Key links to the Skill table. |
| Evaluation | Evaluation\_date | DATE | Date on which evaluation is completed |
| Evaluation | Evaluation\_score | DECIMAL(1) | Each skill is evaluated using a score of 0 through 9, 0 being the very beginner and 9 being proficient. Therefore, I assigned the score to DECIMAL type I digit long. This way, different scores for the same skill and player may be compared. |
| Skill | Skill\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Skill | Skill\_description | VARCHAR (32) | Brief description of the skill that the coach can evaluate every practice. Therefore the type is VARCHAR up to 32 characters. |
| Participation | Particip\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Participation | Practice\_id | DECIMAL(12) | Each participation refers to a particular practice, therefore this Foreign Key links to Practice Table. |
| Participation | Player\_id | DECIMAL(12) | Each participation refers to a particular player, therefore this Foreign Key links to Table Player. |
| Practice | Practice\_id | DECIMAL(12) | This is a synthetic primary key, so the use of decimal type is appropriate |
| Practice | Practice\_date | DATE | Each practice takes place on a certain date. |
|  |  |  |  |

1. **Normalization**

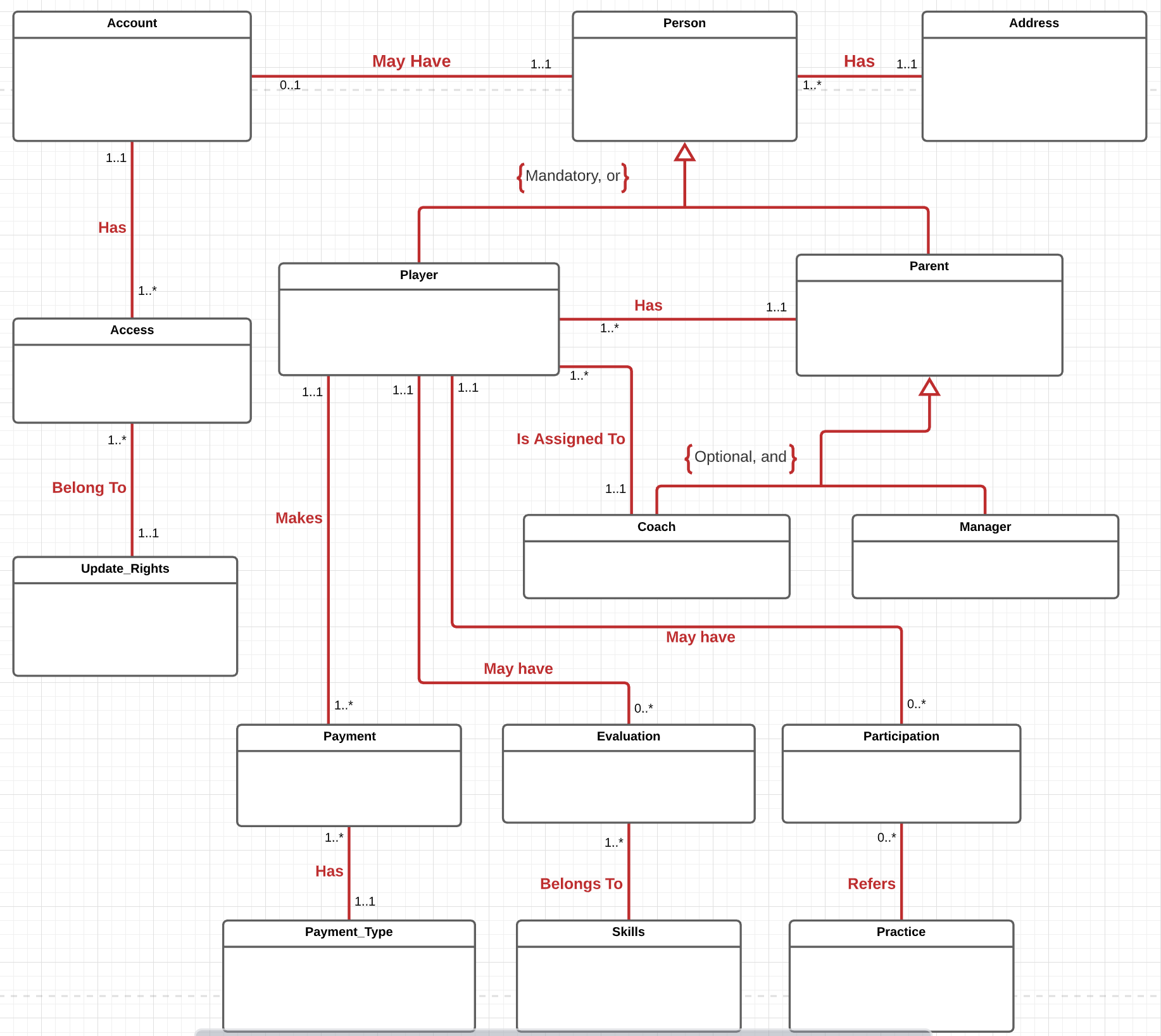
After examining my entities, I realized that the Person has uncontrolled redundancy. All the attributes related to address might have to be repeatedly entered if there are more than one person that shares the same address. Consider the situation in which a parent has several children in the club. Values for the same address will have to be entered several times thus creating redundancy and opportunity for data anomaly. I decided to create a separate table Address which Person will simply refer to.

Table address can also be further separated into several tables for zip and state; however, considering that all players in a given club come from the same state and same town, it is likely that creation of those tables will simply result in extra complexity without any real benefit. Therefore, I decided not the separate zip and state but to keep them all together in Address table.

Separating Address as another entity gives us another structural rule:

A person has one address. Each address belongs to one or more persons.

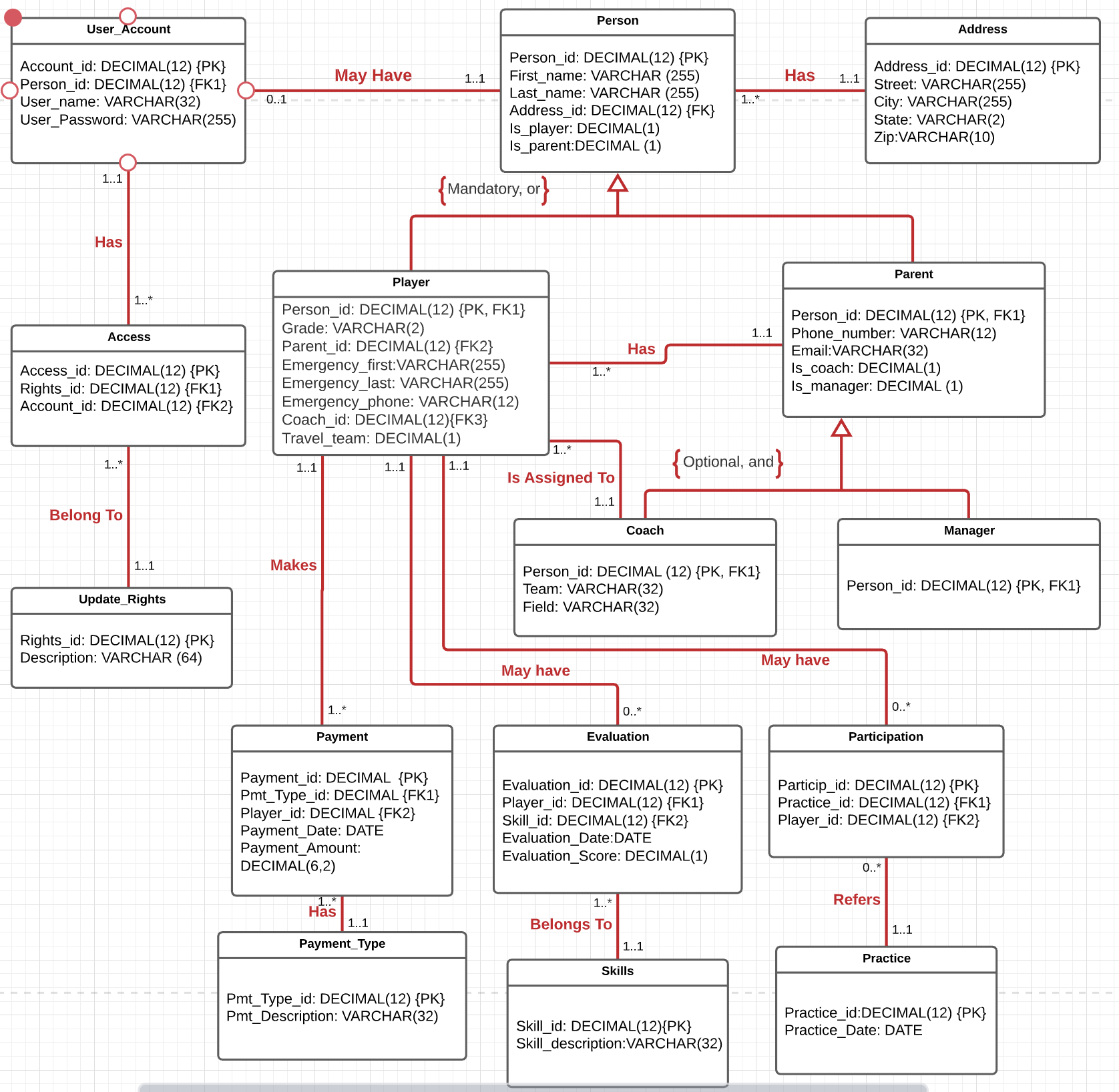
**Figure 2. Conceptual Model ERD after Normalization**



**Physical Model ERD**

Now that I have outlined all the entities, their primary and foreign keys and attributes, I can prepare a **Physical Model ERD.** Please note that both Address and Player are not in BCNF. However, for the sake of simplicity and convenience, I decided to keep those tables as they are.

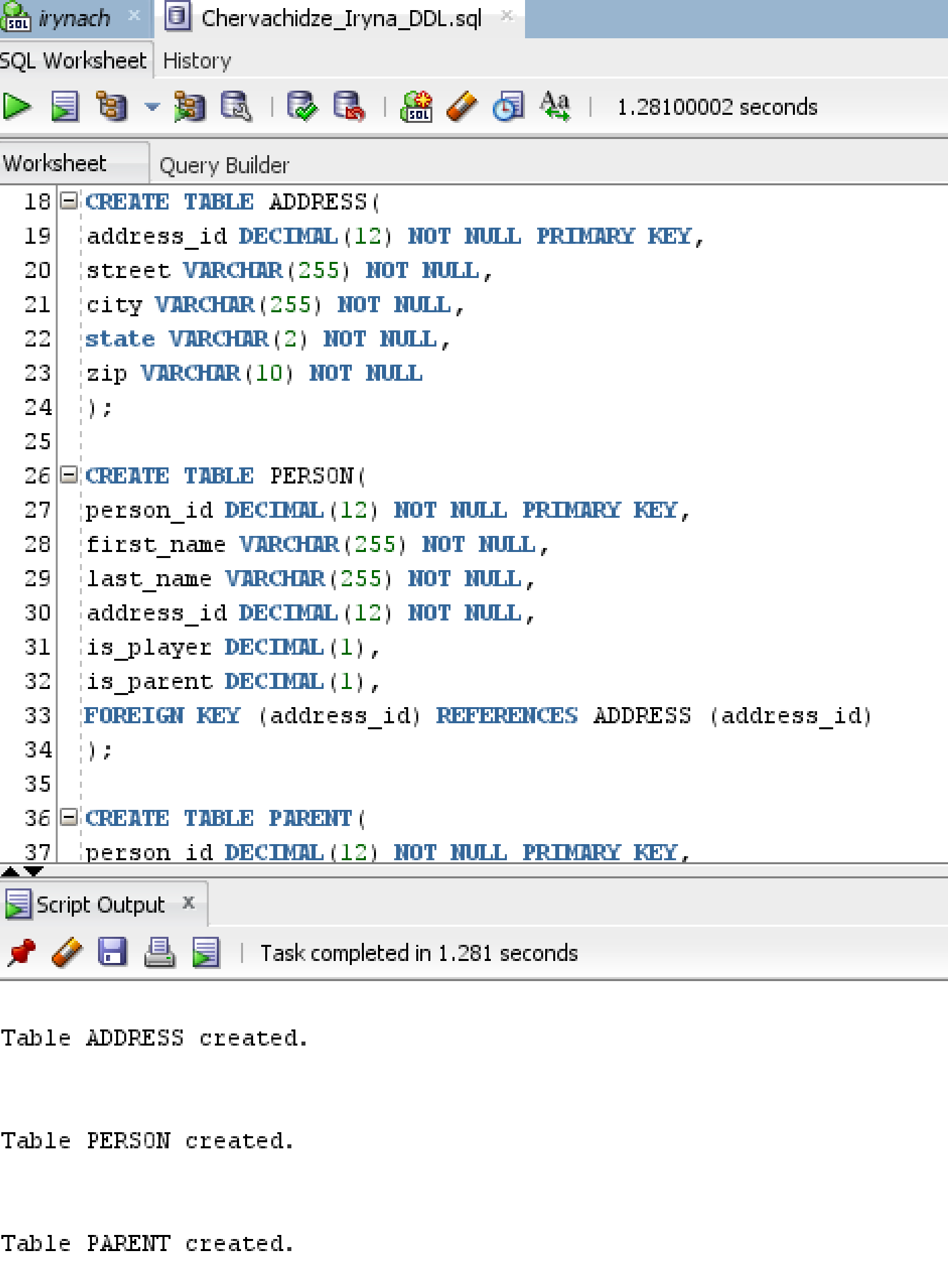
**Figure 3. Physical Model EERD**



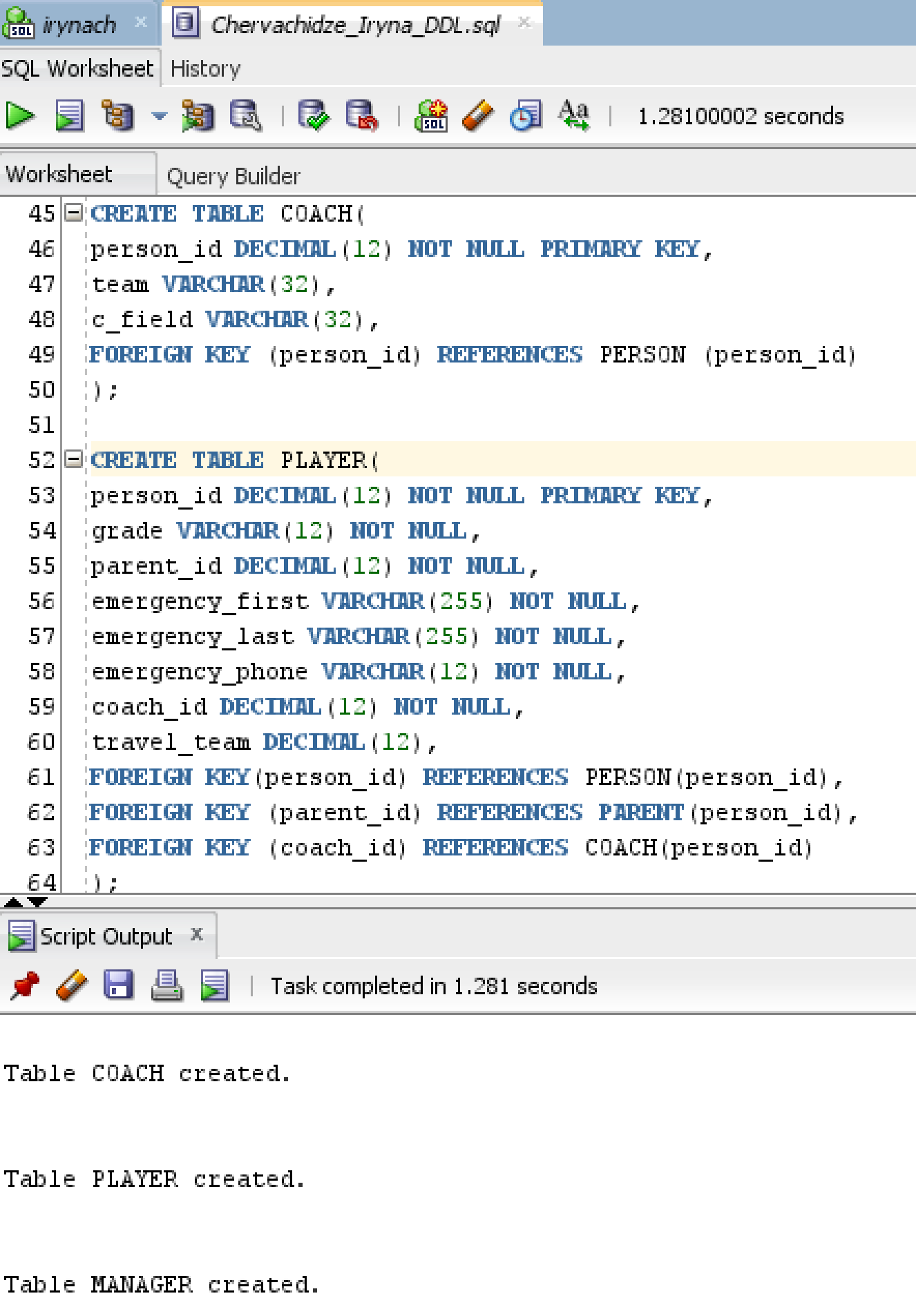
1. **Table creation**

Now that my database EERD is ready, I can create tables in SQL. Here are some sample screenshots of the DDL queries. Full DDL/DML script is included in the appendix.

**Figure 4. SQL Query Creating Tables**



**Figure 5. SQL Query Creating Tables (another example)**



1. **INDEXES**

Creation of indexes is an integral part of database design. We know that primary keys are automatically indexed by the database. The following is the list of all primary keys:

Person.Person\_id

Address.Address\_id

Account.Account\_id

Access.Access\_id

Update\_Rights.rights\_id

Payment.Payment\_id

Payment\_type.Pmt\_type\_id

Evaluation.Evaluation\_id

Skills.Skill\_id

Participation.Particip\_id

Practice.Practice\_id

Foreign keys all need indexes. Therefore, I list all foreign keys with their description in the following table:

**Table 1. Foreign Key Indexes Description**

|  |  |  |
| --- | --- | --- |
| **Foreign Key** | **Unique?** | **Description** |
| Person.Address\_id | Not unique | Because several people may reside at the same address, like a parent and a player, this index cannot be unique |
| Account.Person\_id | Unique | Because each account may only be linked to one person, this index is unique |
| Access.Rights\_id | Non-unique | Because same rights may be shared by several coaches/managers, this index is not unique |
| Access.Account\_id | Non-unique | Each account can have many accesses, therefore this index cannot be unique |
| Player.parent\_id | Non-unique | Each parent may have more than one child in the club, therefore this index cannot be unique |
| Player.coach\_id | Non-unique | Each coach has one or more players in his team, therefore this index is not unique. |
| Payment.player\_id | Non-unique | Because each player may make many payments, this index is not unique |
| Payment.pmt\_type\_id | Non-unique | Each payment type may be made several times (by different players), therefore this index is not unique |
| Evaluation.player\_id | Non-unique | Because each player might have many evaluations (such as for each skill at each practice), this index is not unique |
| Evaluation.skill\_id | Non-unique | Each skills is evaluated multiple times (for different players, as an example) this index cannot be unique |
| Participation.player\_id | Non-unique | Because each player can have many participations, this index is not unique |
| Participation.practice\_id | Non-unique | Because multiple evaluations are done for each practice, this index is not unique |

After analyzing the Physical ERD, I also think the following attributes may benefit from indexes:

Person.First\_name

Person.Last\_name

Player.emergency\_phone

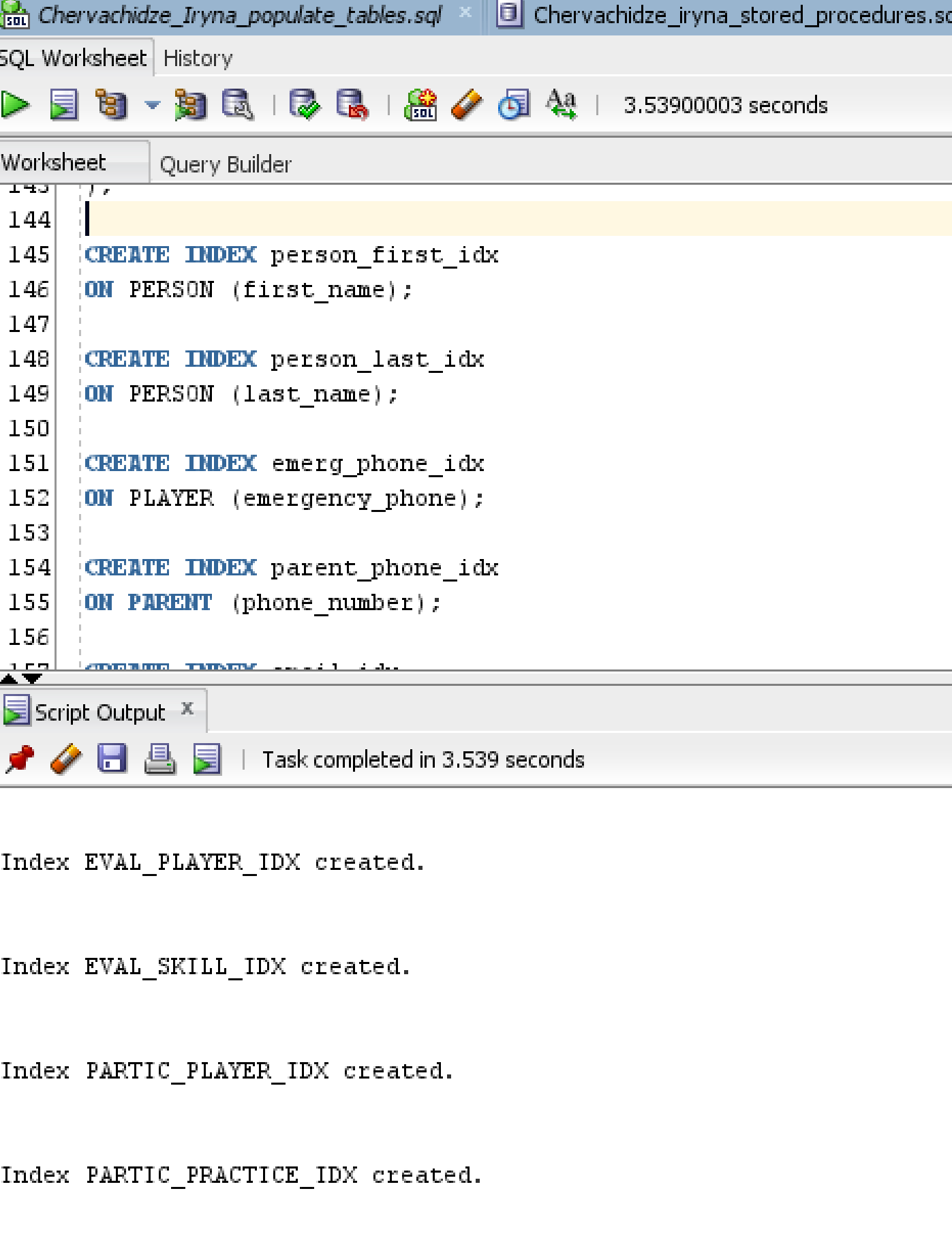
Parent.phone\_number

Parent. Email

Payment.Payment\_date

Practice.practice\_date

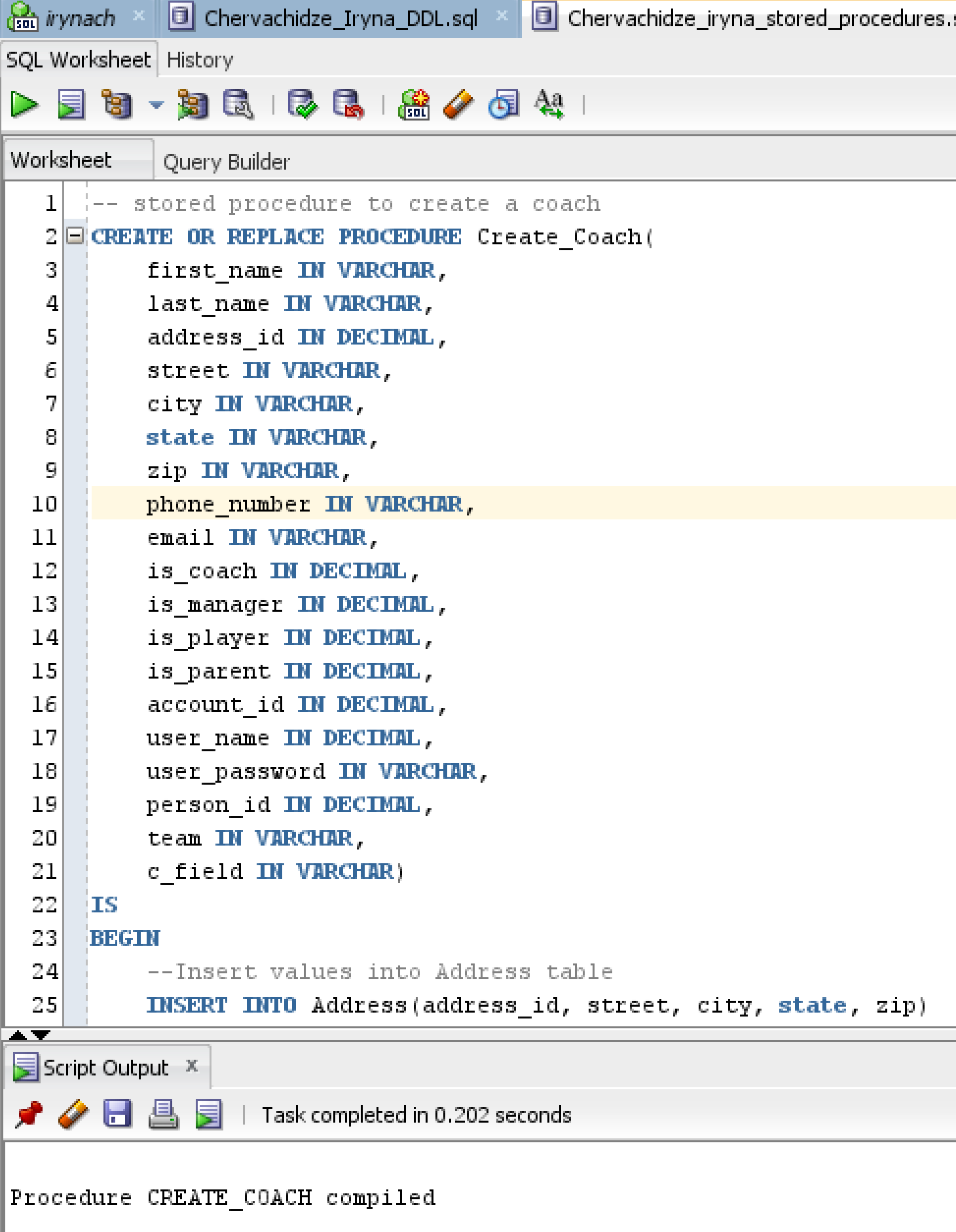
**Figure 6. SQL for Creation of Indexes**

****

1. **STORED PROCEDURES**

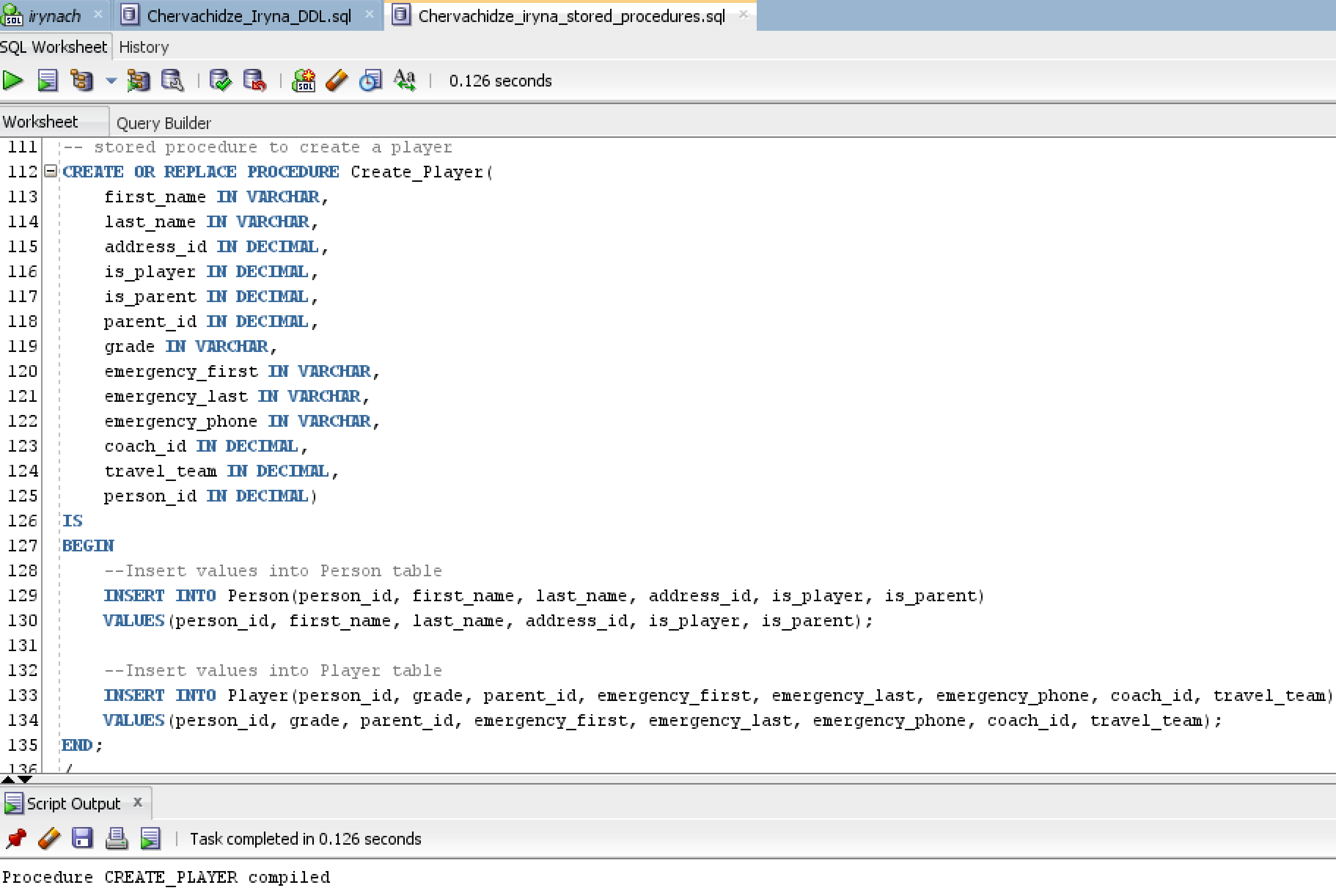
I created several stored procedures for creating the following: an instance of a Person, a Parent, a Player, a Coach, a Payment and Skills. Let’s look at one of the procedures, Create\_Coach, which creates an instance of a Coach. This procedure involves inserting values into multiple tables. When a coach first signs up for an account, he must enter a lot of personal information. This information is then fed into the procedure as parameters and subsequently recorded in multiple tables. To create one instance of Coach, several tables are involved: Address, Person, User\_Account, Parent, and Coach.

**Figure 7. SQL Query Creating a Stored Procedure (Coach Table)**



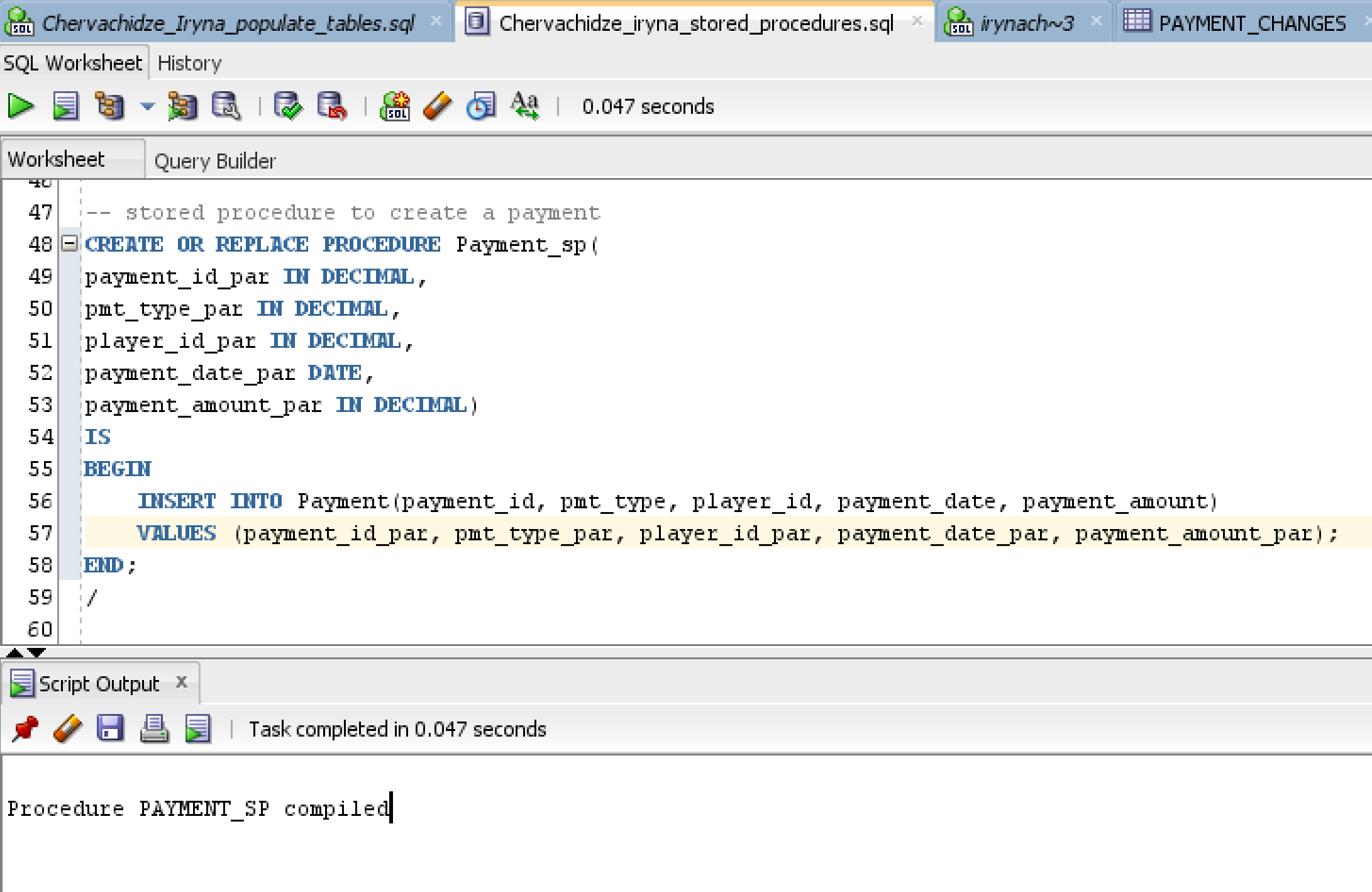
Another stored procedure Create\_Player performs similar tasks. To create one instance of a player, two tables require insertion of values: Person and Player. Address values may be retrieved from the parent of the player though the use of parent\_id foreign key. Here is the screenshot of the Create\_Player stored procedure.

**Figure 8. SQL Query Creating Stored Procedure (Create Player Table)**



The last procedure creates an instance of Payment. This is the simplest of the three, as it only requires insertion of values into one table.

**Figure 9. SQL for Creating Stored Procedure Payment\_sp**

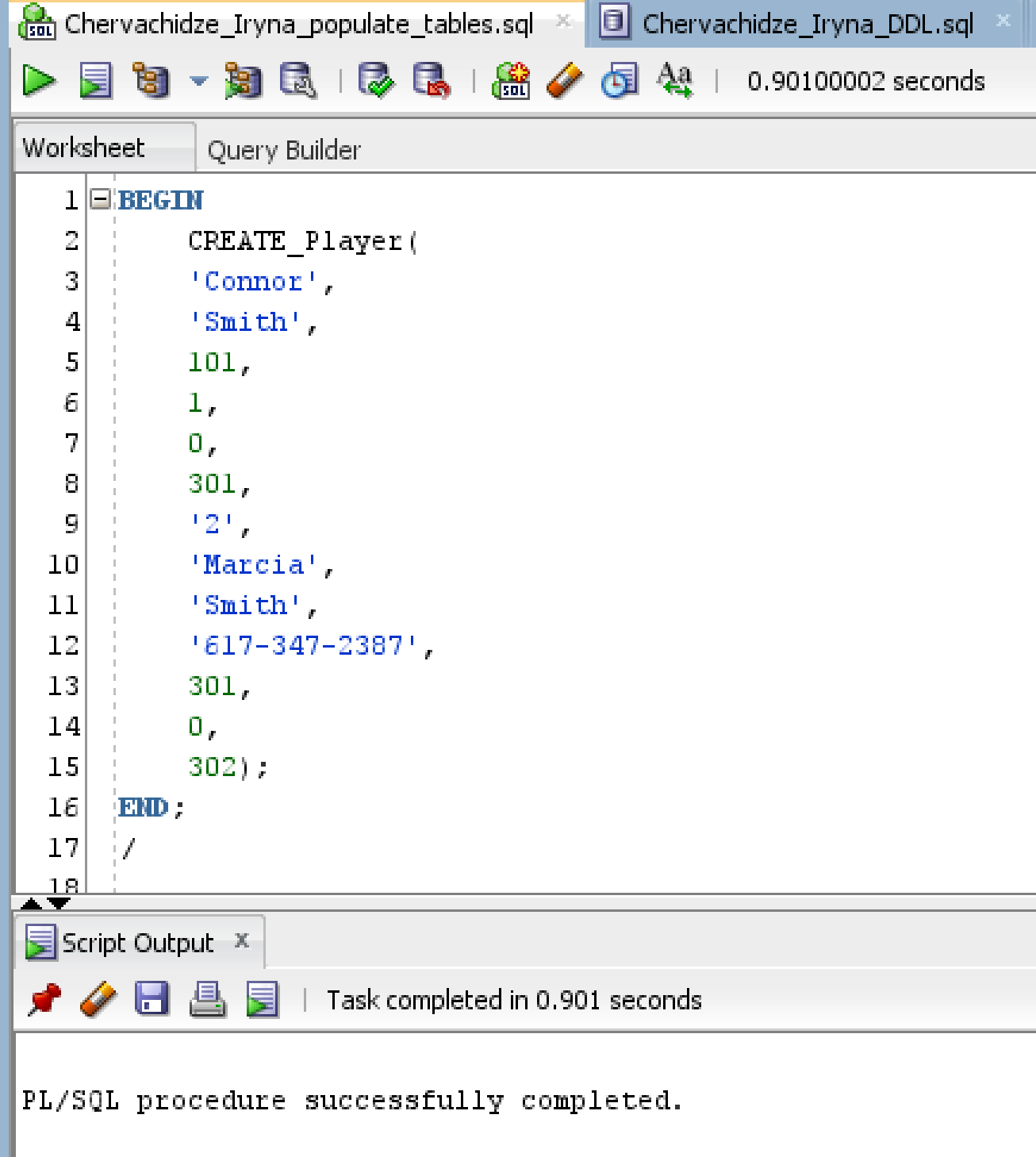


Now that the procedures are compiled and ready, I can start populating my tables using stored procedures. The first one creates an instance of Coach:

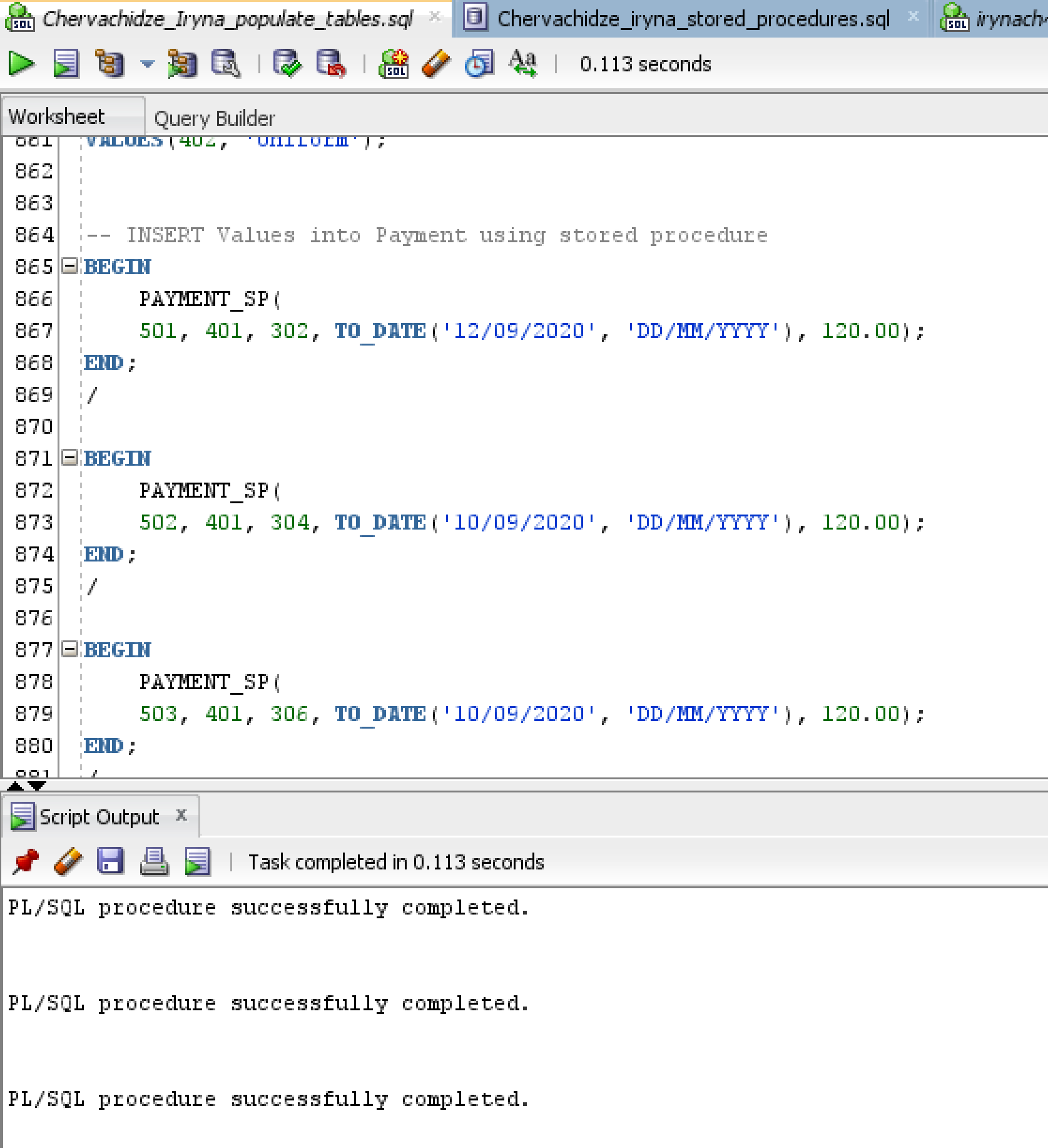
**Figure 10. SQL Query Using Stored Procedure to Create a Coach**



**Figure 11.SQL Creating an Instance of Player Using Stored Procedure**



**Figure 12. SQL for Creating Instances of Payment Using Stored Procedure**



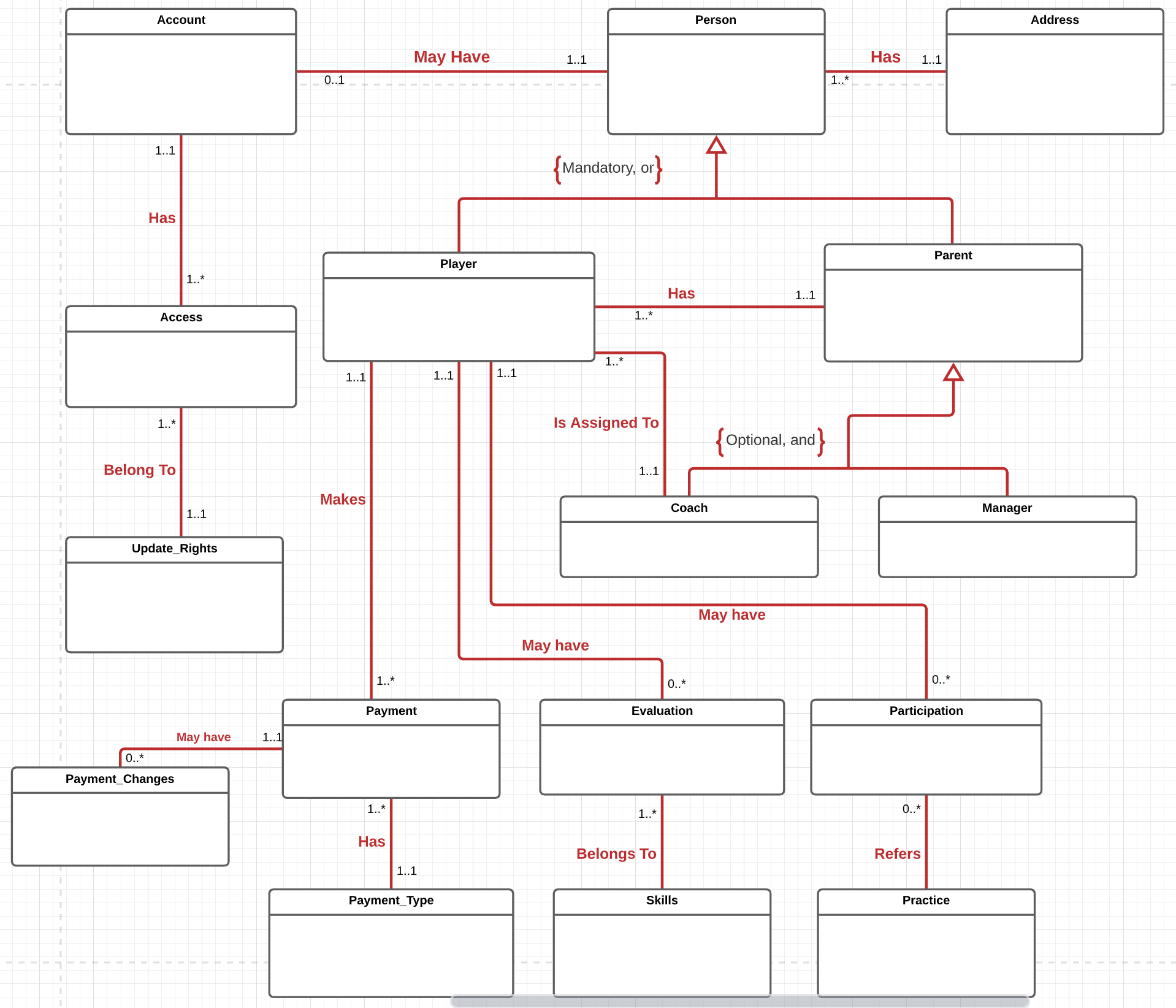
1. **TYS HISTORY TABLE WITH A TRIGGER**

After examining my EERD, one table stands out that may be a good candidate for a history tracking: Payment. I can record changes of the annual fees and prices of the uniform in a separate entity Payment\_Changes. Here is a new structural rule that involves this entity:

Each payment may have many Payment\_Changes. Each instance of Payment\_Changes is related to one Payment.

Having another entity and an additional rule brings some changes to the design of my database. Here is a new conceptual ERD:

**Figure 13. Conceptual ERD After Adding History (Payment Changes)**

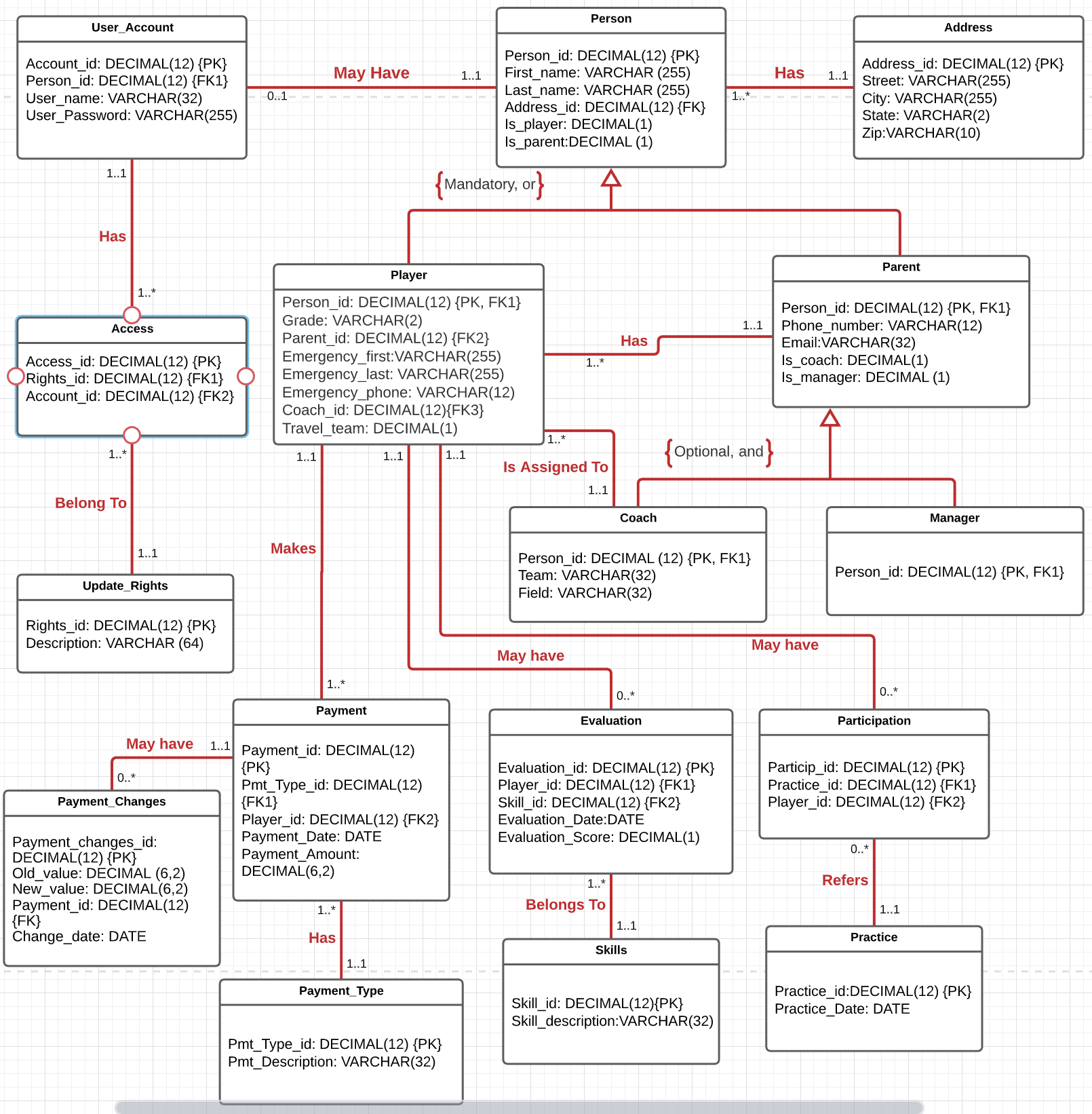


Payment\_Changes table was added to the ERD above. In the following table I list attributes for Payment\_Changes.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| Payment\_changes\_id | Primary key for this entity. Type: DECIMAL(12) as usual for all other primary keys. |
| Old\_value | The value before the change. Type: same as Payment.payment\_amount type, DECIMAL (6,2) |
| New\_value | The value after the change. Type: same as Payment.payment\_amount type, DECIMAL (6,2) |
| Payment\_id | A foreign key that references the instance in the Payment table that had the change in value. Type: same as primary key in Payment table, DECIMAL(12) |
| Change\_date | Date when the value change took place. Type: DATE |

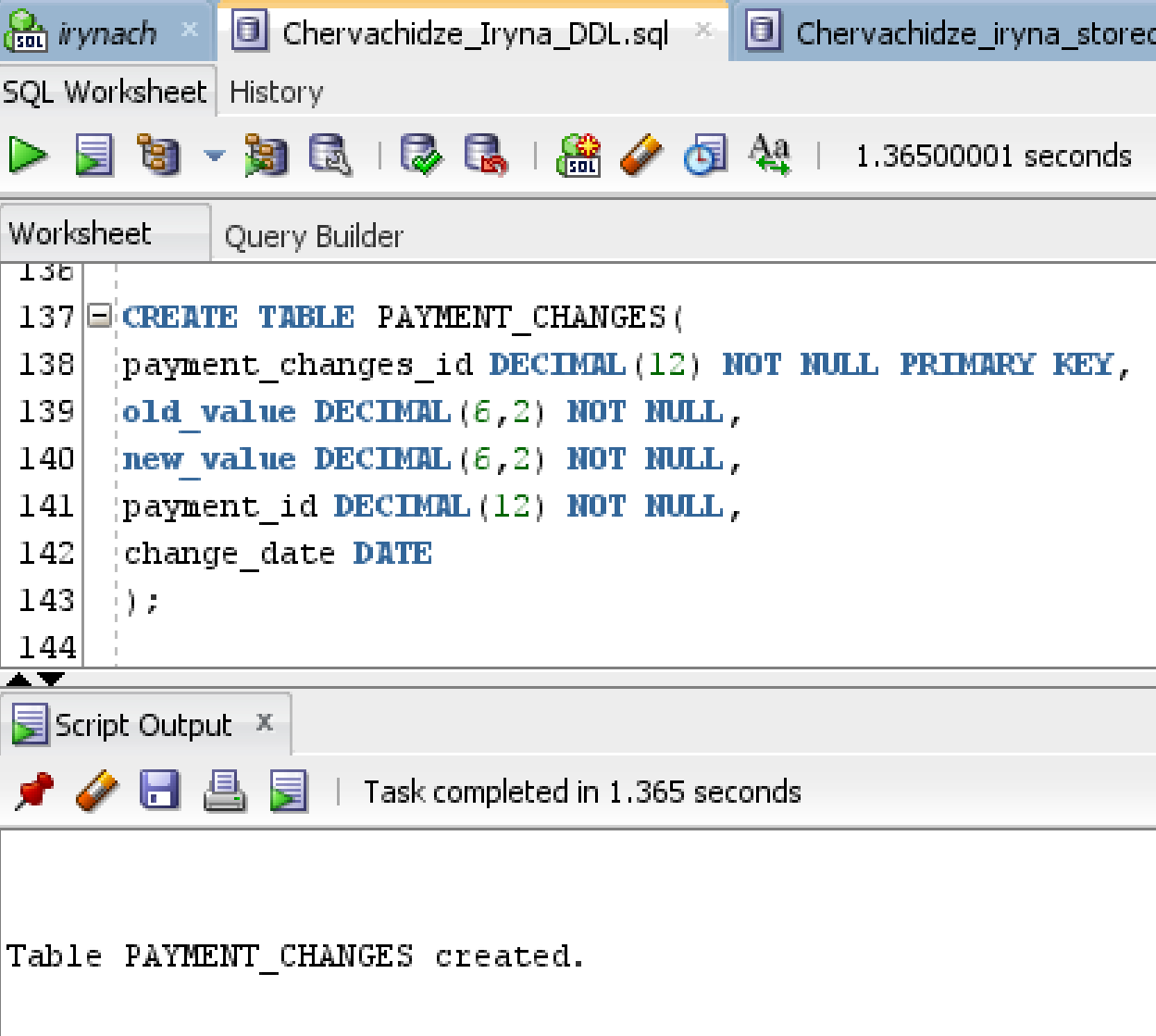
Now the physical EERD looks like the following:

**Figure 14. Physical EERD After Adding History (Payment\_Changes)**



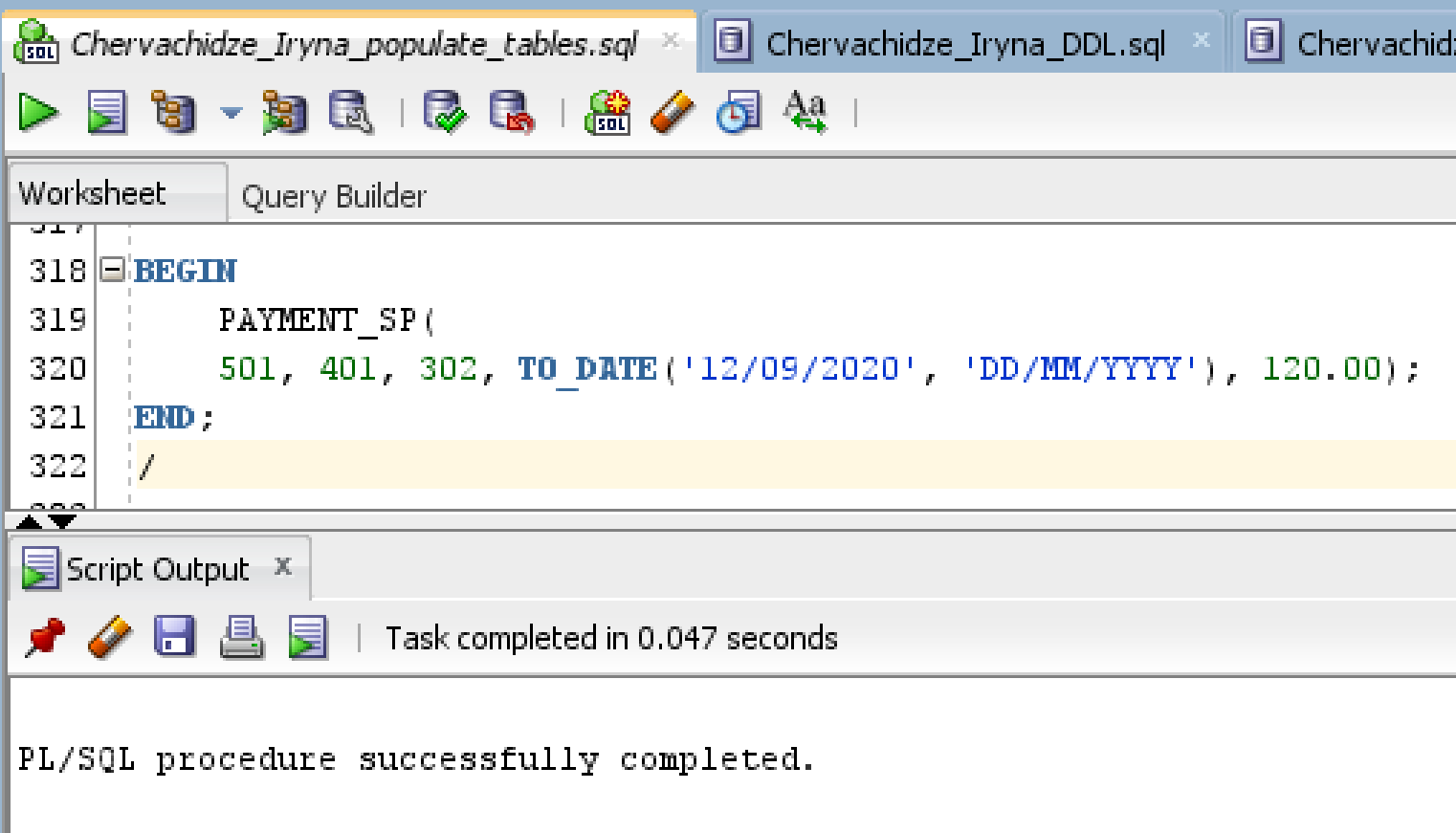
The following is the screenshot of the script that creates table Payment\_Changes. The table has all the attributes listed above.

**Figure 15. SQL Query Creating Table Payment\_Changes**

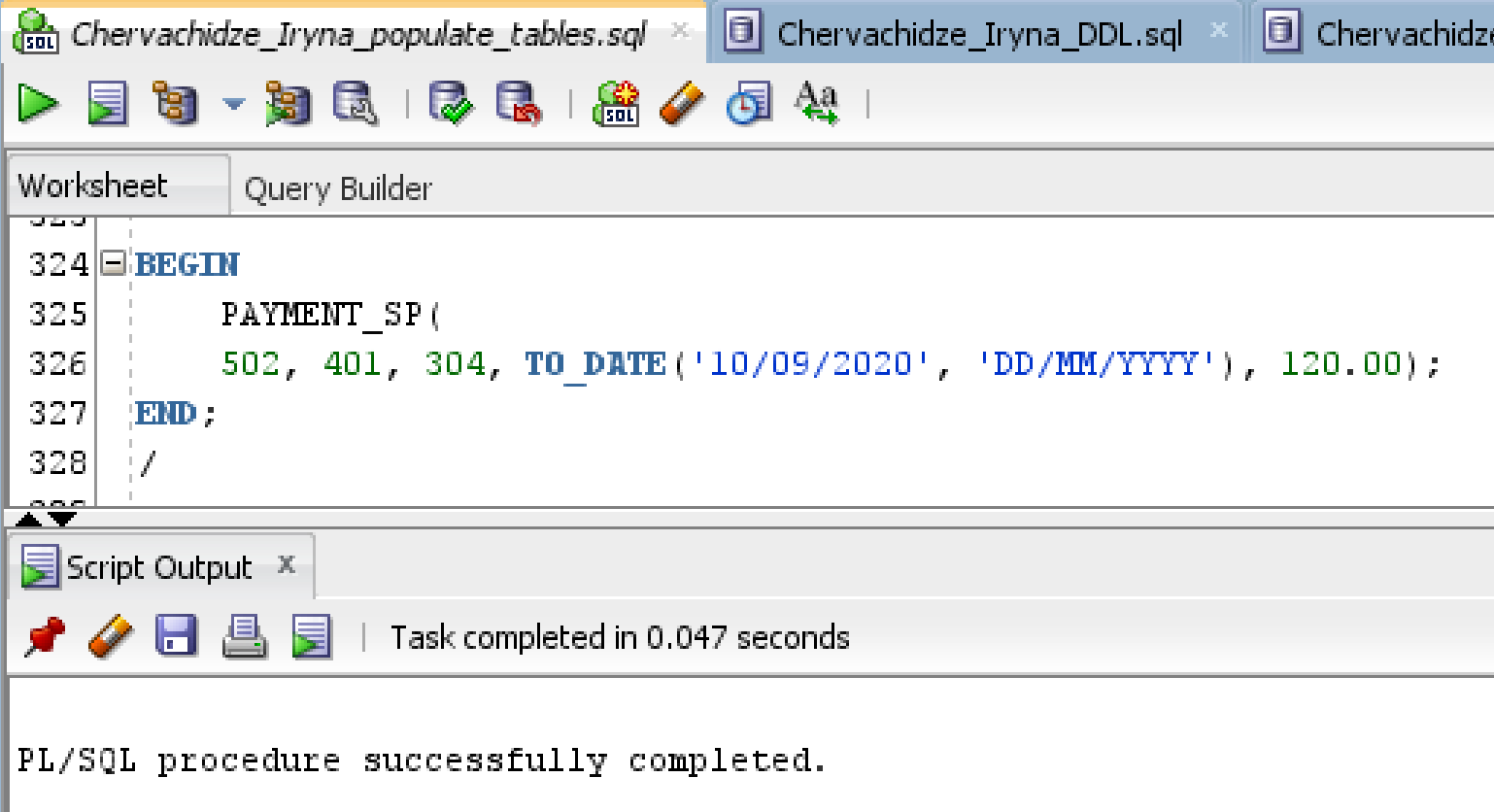


Now that the table is created, I populate it with data so that I can later change the payment amount and test if my trigger procedure works as expected.

**Figure 16.SQL Query to Populate Table Payment(a)**

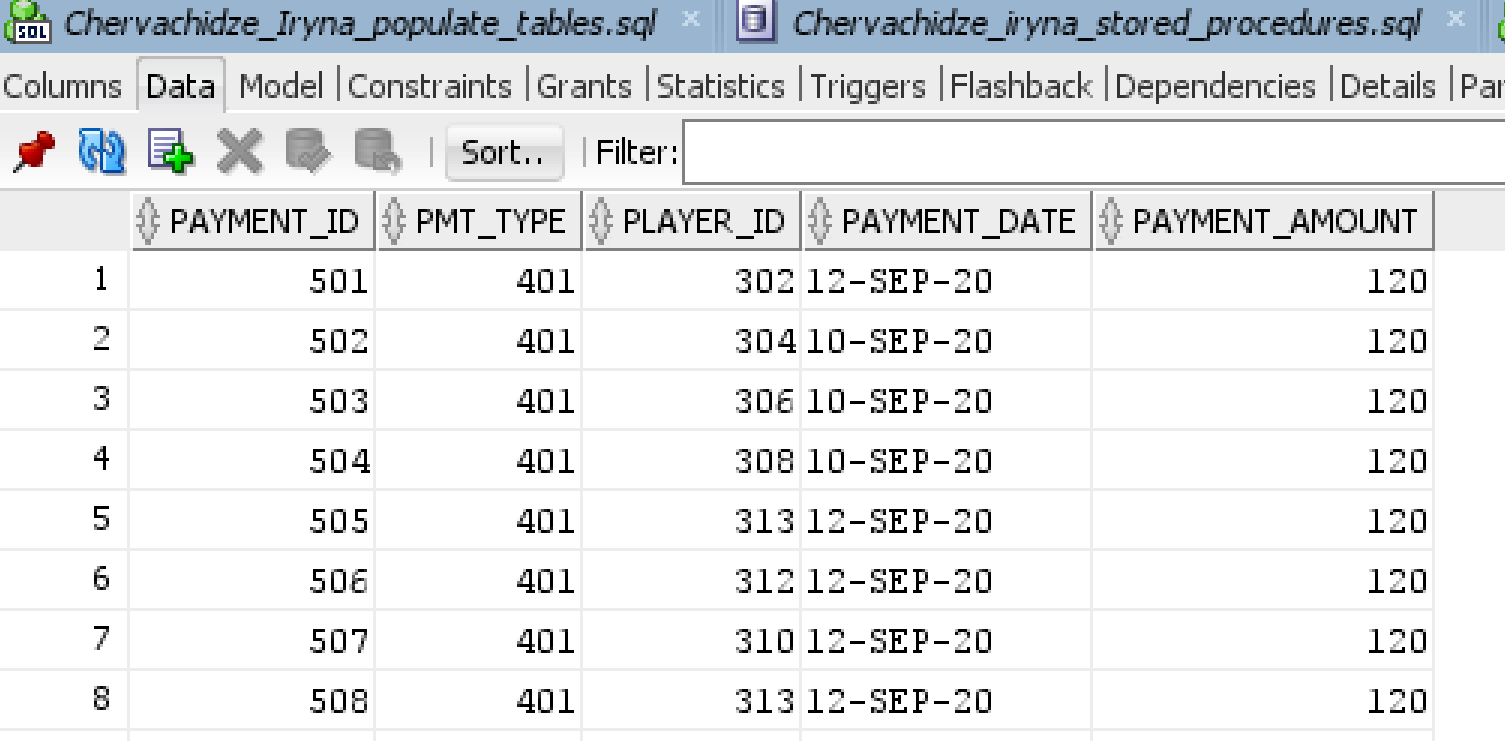


**Figure 17.SQL Query to Populate Table Payment(b)**



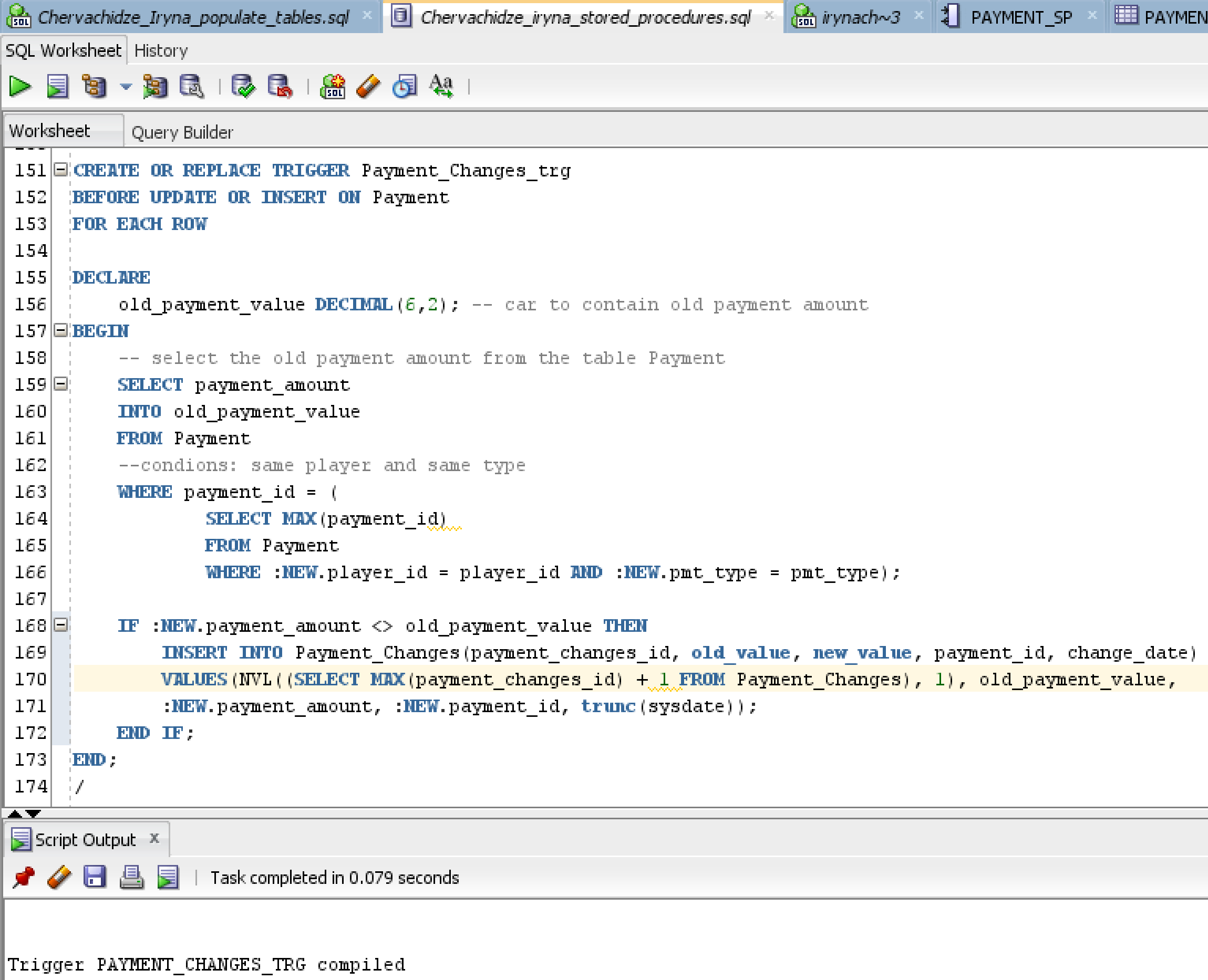
Here is how the Payment table looks like before I test the trigger:

**Figure 18.Table Payment**



Here is the SQL script for the Trigger that populates Payment\_Changes table when the changes in payments happen.

**Figure 19.SQL Query Creating a Trigger to Populate Payment\_Changes table**



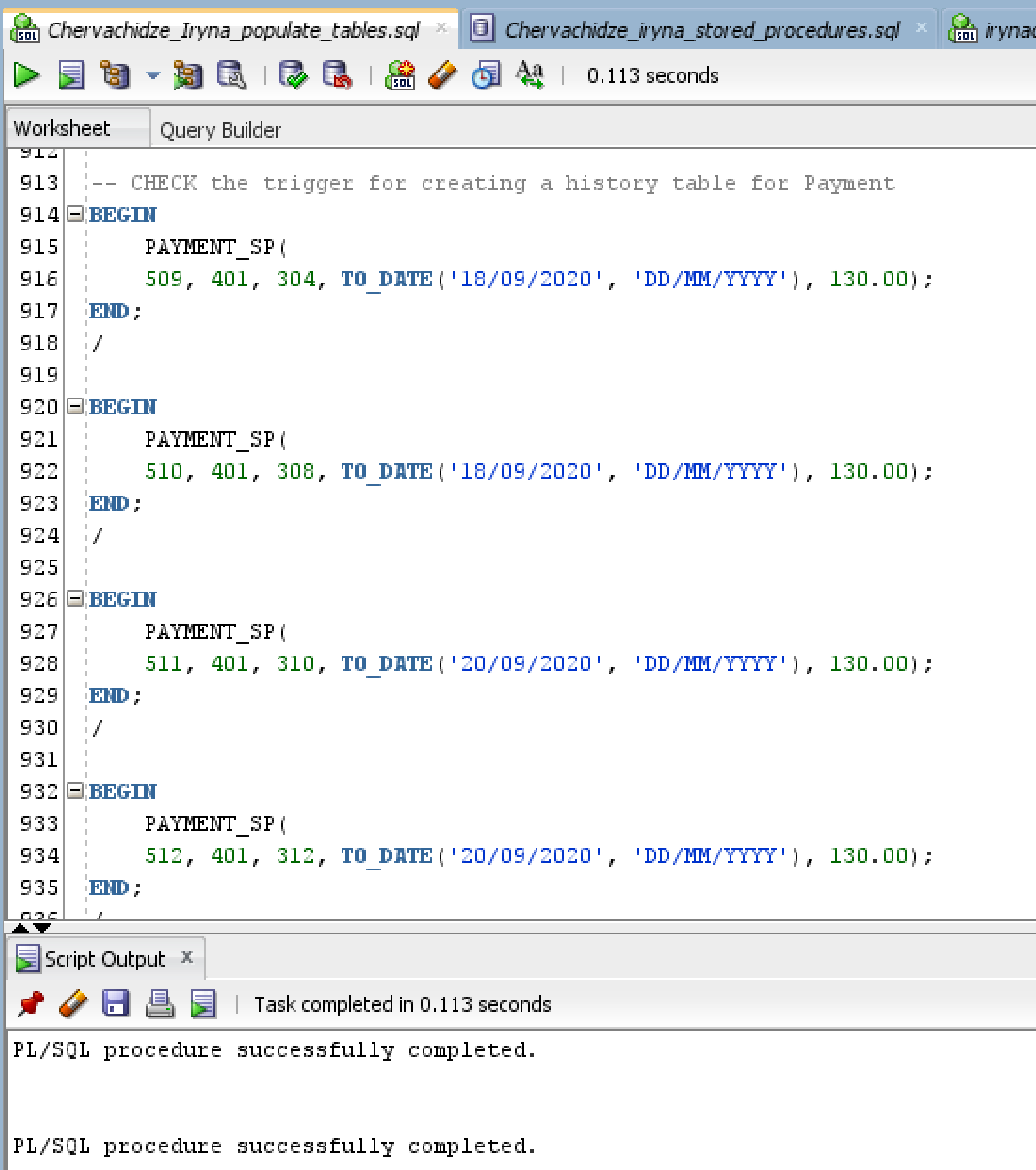
**Table 2. Trigger Logic Explanation**

|  |  |
| --- | --- |
| **CREATE OR REPLACE TRIGGER Payment\_Changes\_trg**  **BEFORE UPDATE OR INSERT ON Payment**  **FOR EACH ROW** | This starts the definition of the trigger named Payment\_Changes\_trg. It is linked to the Payment table and is executted before the insertion or update of that table. |
| **DECLARE**  **old\_payment\_value DECIMAL(6,2); -- car to contain old payment amount** | Declaring a variable to contain the value of the old payment, for comparision and for inserting into the table Payment\_Changes |
| **BEGIN** | Part of syntax |
| **SELECT payment\_amount**  **INTO old\_payment\_value**  **FROM Payment**  **--condions: same player and same type**  **WHERE payment\_id = (**  **SELECT MAX(payment\_id)**  **FROM Payment**  **WHERE :NEW.player\_id = player\_id AND :NEW.pmt\_type = pmt\_type);** | This is a query that selects the value of the old payment. It is important that we compare new value and old value for the same player and the same type of payment. In the subquery I select the most recent value of the payment by choosing the max of the primary key value. |
| **IF :NEW.payment\_amount <> old\_payment\_value THEN** | Conditional Statement. If it returns true, then the rest of the trigger logic will continue |
| **INSERT INTO Payment\_Changes(payment\_changes\_id, old\_value, new\_value, payment\_id, change\_date)**  **VALUES(NVL((SELECT MAX(payment\_changes\_id) + 1 FROM Payment\_Changes), 1), old\_payment\_value,**  **:NEW.payment\_amount, :NEW.payment\_id, trunc(sysdate));** | This is the regular insert statement. If the condition above is satisfied, then the trigger inserts all these values into the table Payment\_Changes. |
| **END IF;**  **END;**  **/** | More of the necessary syntax |

Now imagine that a year passed and a new season begins. The new annual fee has changed from $120 to $130. I am going to insert a new payment with this new changed fee for one of the players and check if the trigger procedure I created will record the change in the Table Payment\_Changes.

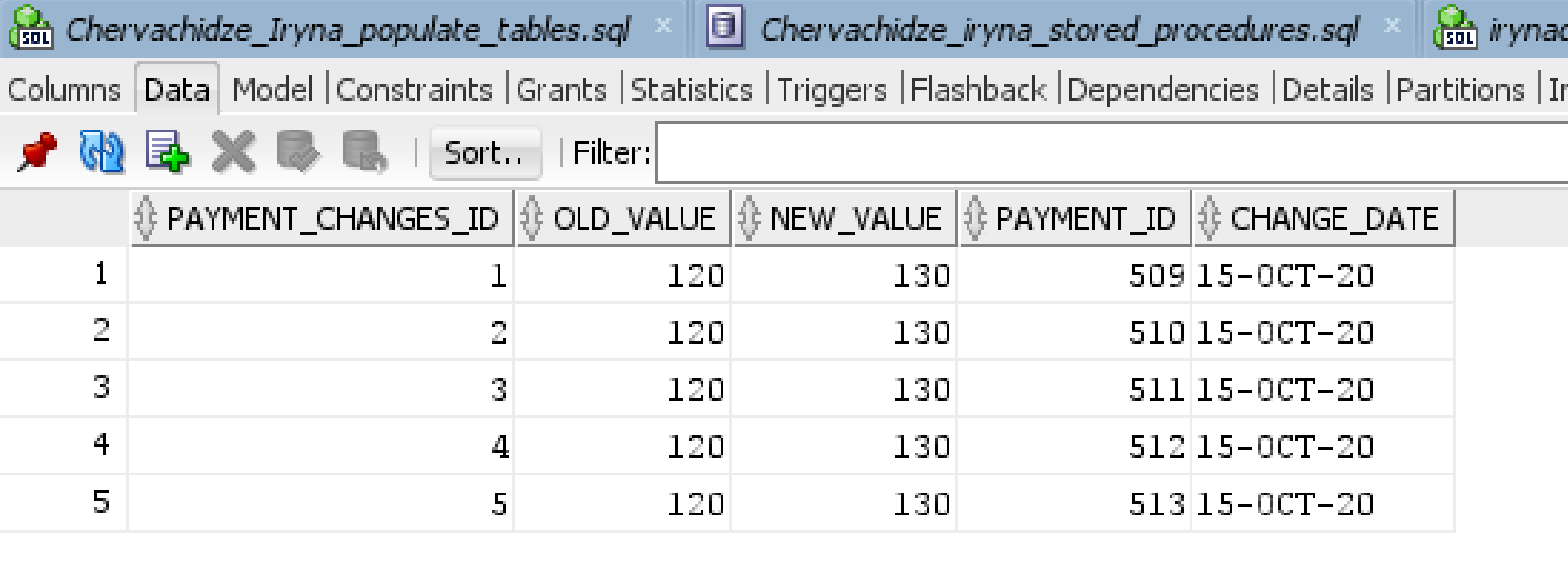
Here, I insert a new payment with an annual fee that increased from $120 to $130 for two of the players.

**Figure 20. SQL Query with a New Annual Fee to Check the Trigger**



Here is the Payment\_Changes table that contains changes in payment values as a result of the trigger detecting a change in the annual fee.

**Figure 21.Table Payment\_Changes**



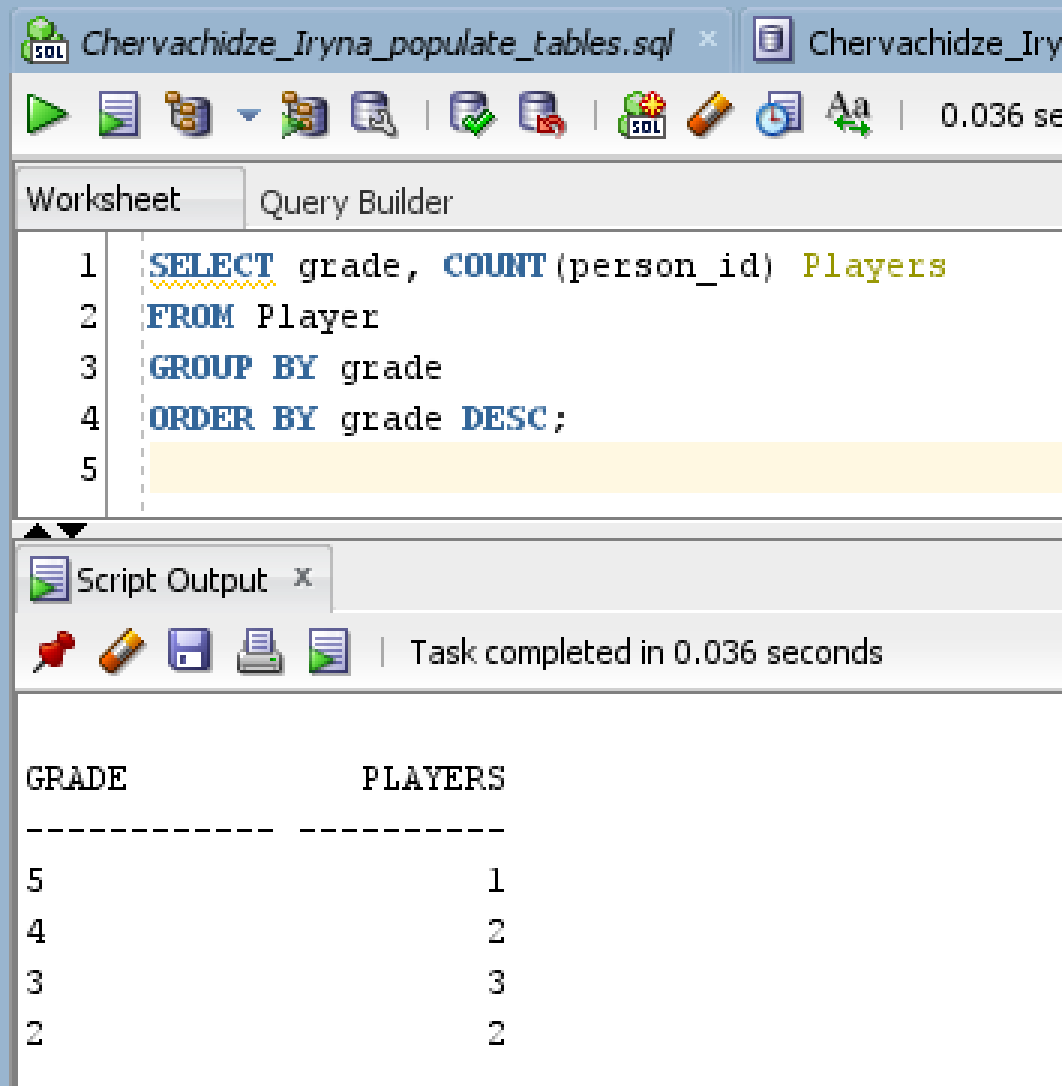
The table Payment\_Changes contains two records that correspond to the two insert statements aobe where the annual fee changed from the last year.

1. **TYS QUESTIONS AND QUERY**

**10.1**

The first question is a simple but a very useful one. Since the players in the town soccer usually practice and play with the other players of the same grade, the club needs to know how many players have enrolled in each grade to make teams. This is a very simple query that uses one aggregate function and one GROUP BY as well as ORDER BY clause.

**Figure 22.SQL Query to Find the Number of Players by Grade**



Below is the table Player with the records sorted by the grade in Descending order. It is easy to see that the results of the query are correct.

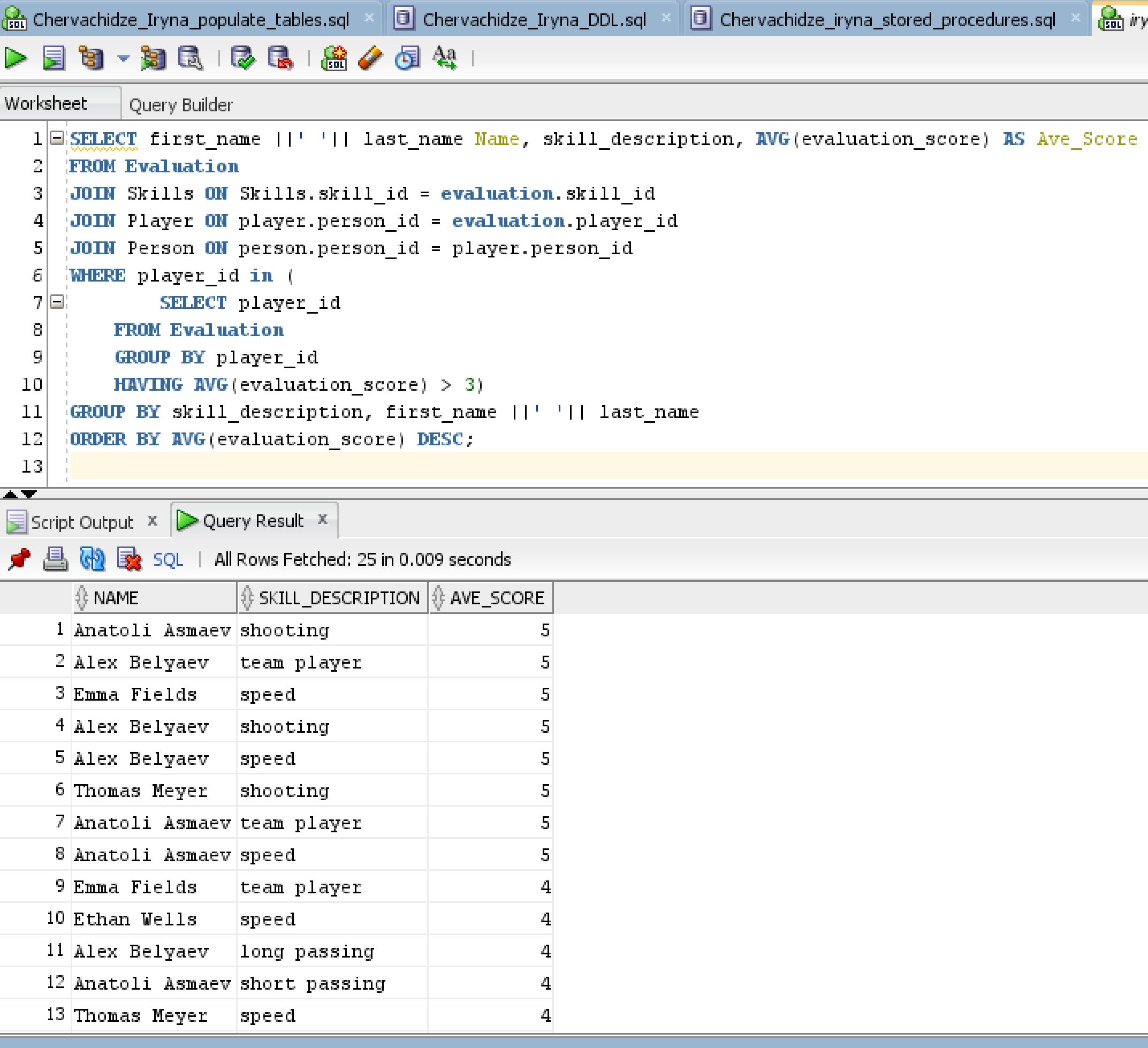


**10.2**

The second question refers to the skill evaluation provided by the coaches to the players. Suppose the club wants to know all players whose skills are more advanced, i.e. who scored 4 or more on average for all skills. It may be useful because the town can then form a team of strong players who would compete against other towns. Therefore, the club wants a list of all players whose total score across all skills is no less then 4 along with the list of all skills per player with the average score for that skill.

SQL Query is provided below:

**Figure 23. SQL Query to Find all Players with Total Average Score no less than 3**



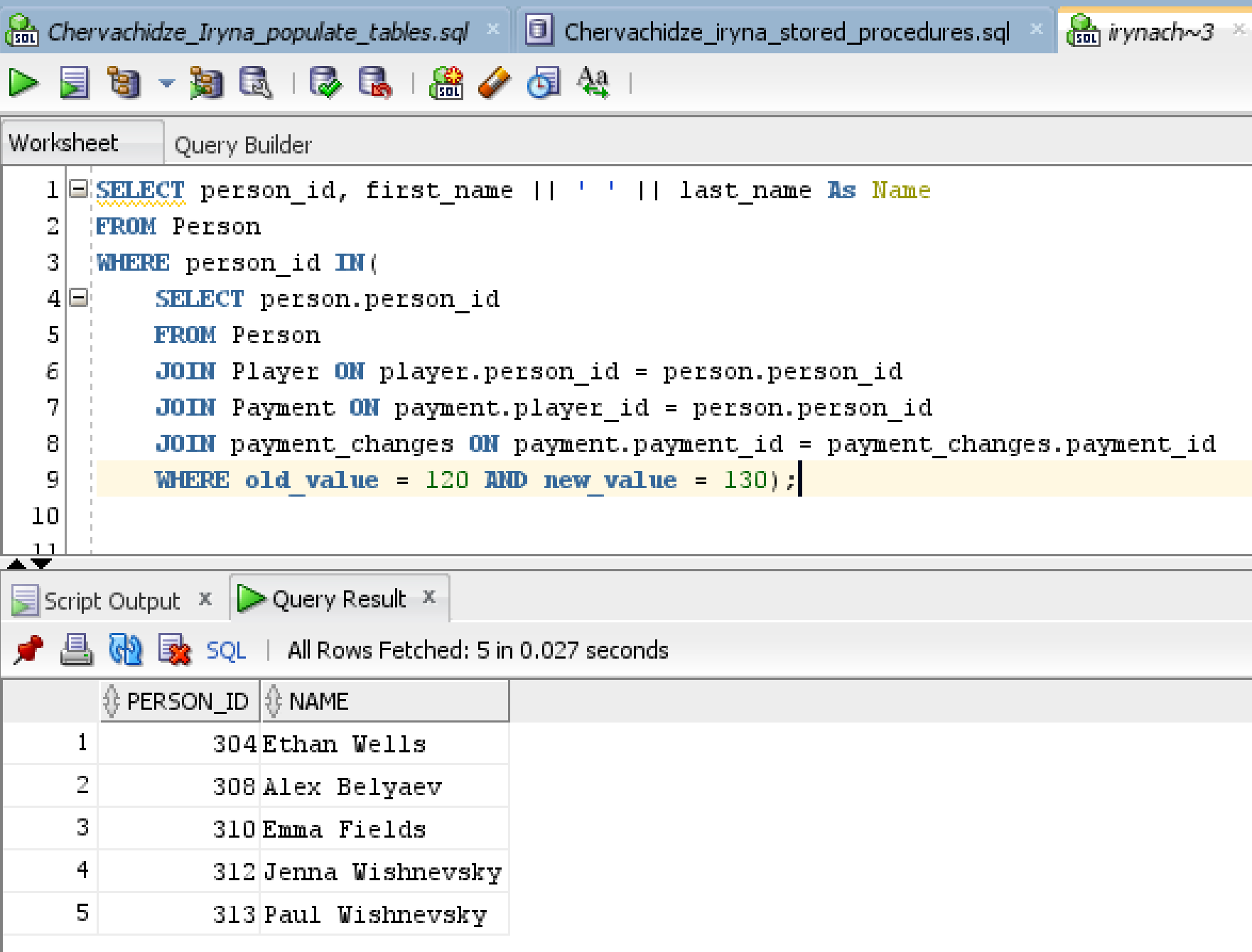
**Table 3. SQL Query Explanation for Finding Player with an Average Score of 4 or more**

|  |  |
| --- | --- |
| **SELECT first\_name ||' '|| last\_name Name, skill\_description, AVG(evaluation\_score) AS Ave\_Score** | The first line just providesthe names of the attributes that we want to see in the result set. Evaluation score is aggregated into averages. |
| **FROM Evaluation**  **JOIN Skills ON Skills.skill\_id = evaluation.skill\_id**  **JOIN Player ON player.person\_id = evaluation.player\_id**  **JOIN Person ON person.person\_id = player.person\_id** | Joins describe what tables we need to join to retrive our result. |
| **WHERE player\_id IN (** | Filtering condition |
| **SELECT player\_id**  **FROM Evaluation**  **GROUP BY player\_id**  **HAVING AVG(evaluation\_score) > 3)** | Subquery specifying what players we need to consider. Since we only need players with the average total score of no less than 4, we group out selection by player and then choose only those whose score is > 3. |
| **GROUP BY skill\_description, first\_name ||' '|| last\_name**  **ORDER BY AVG(evaluation\_score) DESC;** | Finally, I group the result by skill and order by the average score for each skill. This way, the coaches will be able to see easily which players possess highest ranking skills. |

**10.3**

The third question involved the use of the history table created earlier. We can easily look up how many payments were made for the previous annual fee and how many were made for the current fee. Since every player needs to pay a fee, then we can conclude that the number of payments is equal to the number of players at that year. The club may be interested to know the names of those players who were enrolled last year and are enrolled this year. It is easy to find out using a payment history table. I can choose to display the names of only those players who paid the fee last year, which also means that they were enrolled last year.

**Figure 24. SQL for Finding Those Players Who were Enrolled Last Year and Now**

****

**Figure 25.SQL Code Explanation for Finding Players that were enrolled last year and now.**

|  |  |
| --- | --- |
| **SELECT person\_id, first\_name || ' ' || last\_name As Name**  **FROM Person** | Simple selection of first and last name using alias |
| **WHERE person\_id IN(** | Filtering condition that limit the number of names to display |
| **SELECT person.person\_id**  **FROM Person**  **JOIN Player ON player.person\_id = person.person\_id**  **JOIN Payment ON payment.player\_id = person.person\_id**  **JOIN payment\_changes ON payment.payment\_id = payment\_changes.payment\_id**  **WHERE old\_value = 120 AND new\_value = 130);** | Subquery that limits person\_id only to those who paid the fee last year and this year, which also means they were enrolled last year and now. |

1. **SUMMARY AND REFLECTION**

My application needs to store data related to the TYS activity. This includes personal information of players, their parents, coaches and managers. It also needs to store some payment information as the players need to pay their annual fee and purchase uniforms. Coaches also need to record their feedback as it related to the players skills and attendance.

The Conceptual model ERD as well as physical model ERD have been updated to reflect additional history table Payment\_changes. ERDs outline the important relationships between the antities in the my model.

All associative and specialization relationships are presented in a ERD diagram, along with all primary/foreign keys and attributes. SQL commands to create all tables and indexes are contained in the SQL script.

Stored procedures have been created and used to populate the tables in the database. A trigger that helps maintains historical records of changes in annual fees was created and used to detect such changes. I also identified three business questions that may be relevant to TYS club and answered them with SQL queries.

My database has taken shape and is now ready for use. I enjoyed the process of designing and implementing a database incrementally. It was very instructive and memorable.

.