Introduction to Database HW4(Final Project) Report

一　組員：0416324胡安鳳　0316323薛世恩　夏頡民

二　貢獻度：各佔三分之一

三

Query2（胡安鳳）:select team\_team\_id,count(player\_player\_id) from team,player where

　　team\_team\_id=player\_team\_id and category='badminton' group by

　　team\_team\_id

（採用文字編輯器：Atom）

這個query是用來查詢對於所有打羽毛球的球隊中，其隊伍中的球員總數。

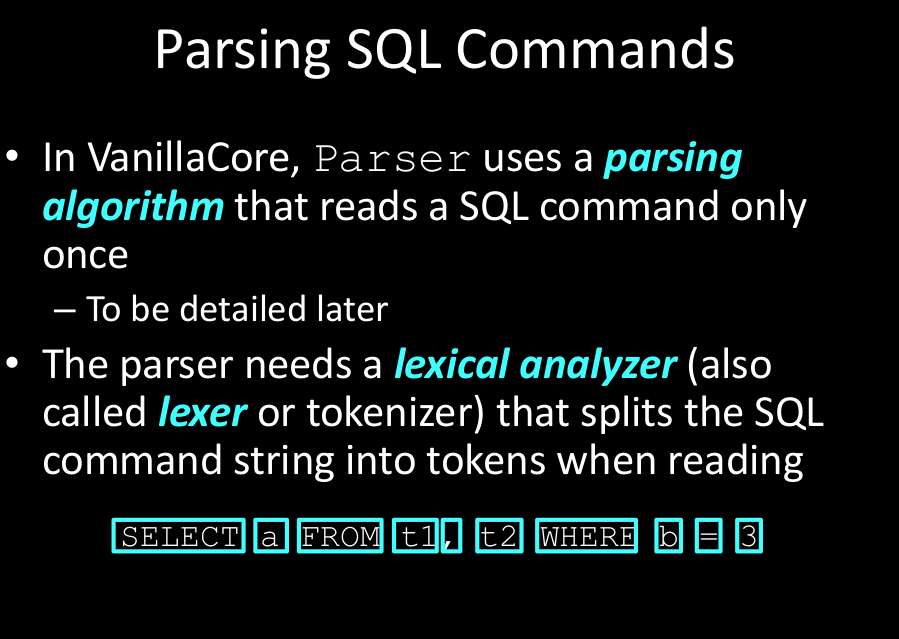
當我們輸入的query其實是被當作一個很長的字串載入SQL processor 中

整個程式的執行順序:

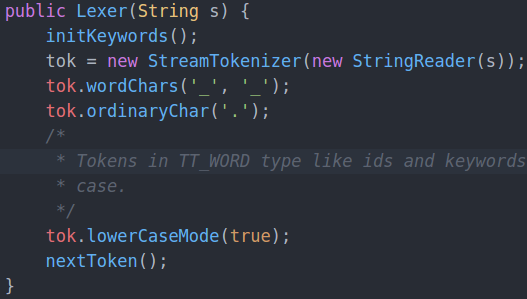
●step1SQL語句分析，確認文法無誤，”Syntax Correctness”

首先由Parser/Lexer執行，其中對於字串的切割是交給Lexer，剖析語意則是交給Parser，兩者相輔相成，如圖中可見Lexer引入了package





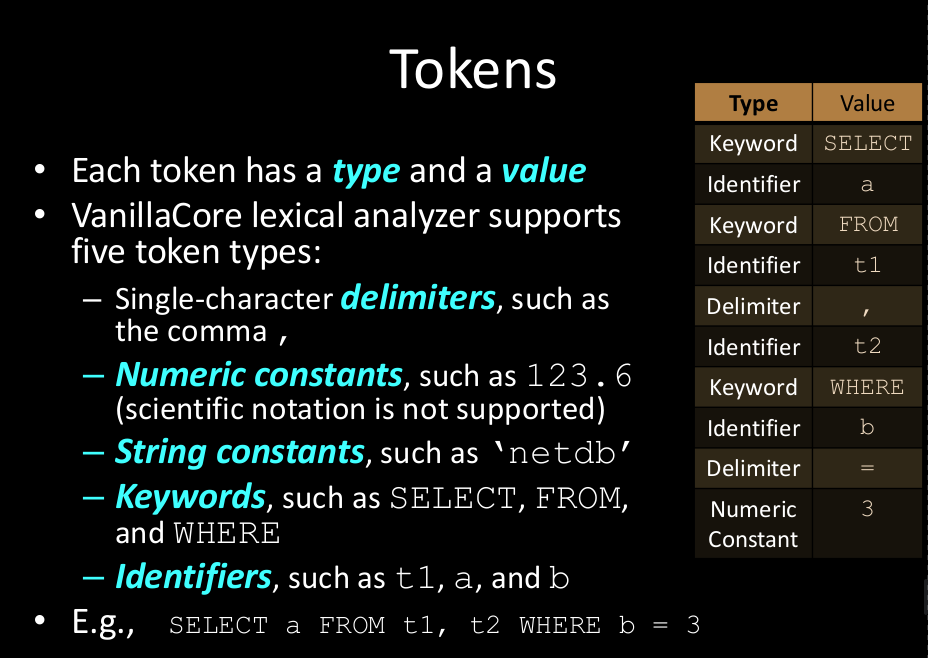
Lexer利用類似c++中StringStream(#include <sstream>)函式庫中切割的方式，同樣都是利用string token(deliminator)這種東西來剖析，如下圖



tok就是用來切割字串的識別符號。其中wordchar這句代表說字詞之間只能用底線連結例如player\_id ，而.當成一般的字符

而不做子查詢(這也是VanillaDB不支援點符號的原因)，其他一律切割。最後將tok一律在小寫的ascii code以免混淆，造成錯誤。

然後以nextToken繼續執行到文末，繼續切割，直到整個輸入的eof(end of file 才會停)



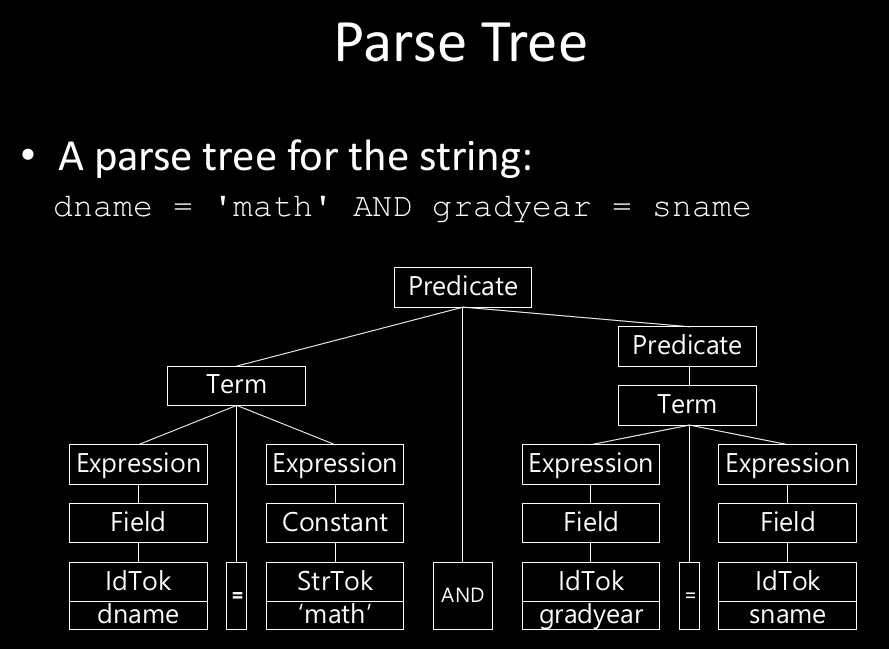
以在同一個Lexer.java中有建立一個字典表，代表會用到的SQL關鍵字

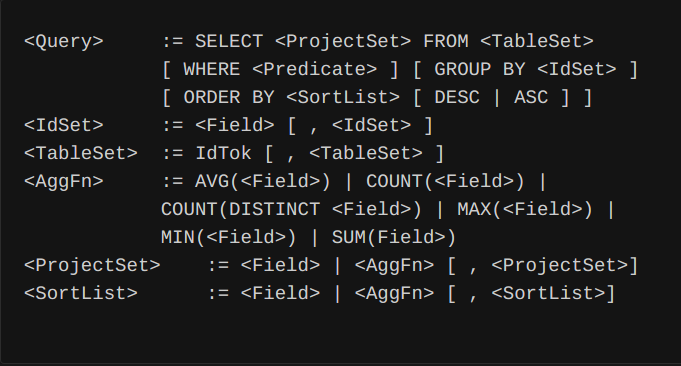
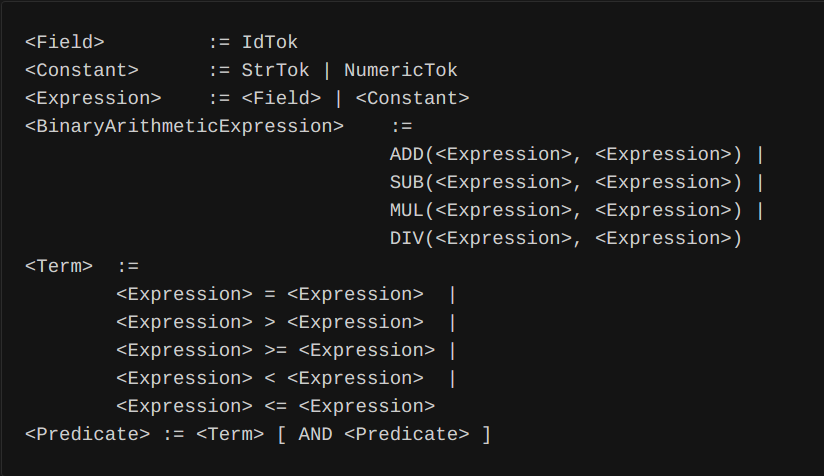


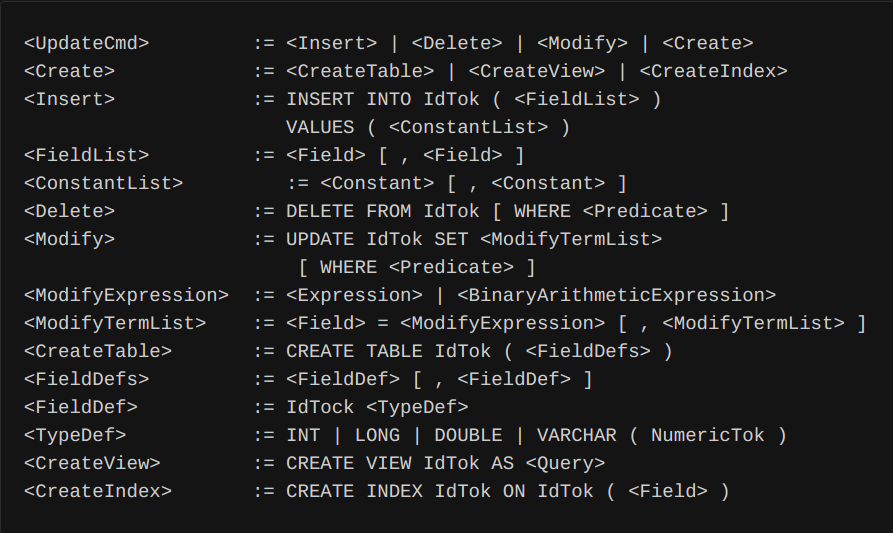
Lexer切完交給Parser後如果有不符合VanillaDBSQL的語法就會報錯，丟出BadSyntaxException這種例外，推斷是運用了『正規語言』的『上下文無關文法』(Context Free Grammar)

用一顆Parsing tree依照SQL的文法剖析輸入的字串

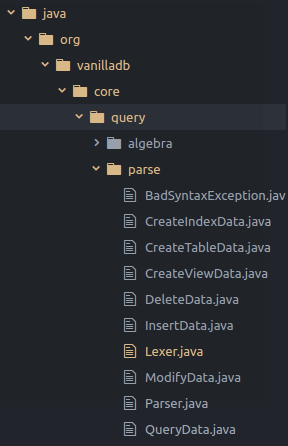
例如:







以上三者分別說明，單一語句(Predicates/左上)，查詢(Queries/右上)，以及更新(Updates/下)的文法規範，在Lexer中便會呼叫Parser中識別輸入的語句是屬於那一種然後找對應以上三者之一的文法歸範(文法剖析樹，如方才提過的Parse Tree)，由以下的檔案樹狀結構照片可以看到，當Lexer呼叫Parser的時候，實際上Parser也會和Lexer互動，由圖片中的檔案結構呼應方才提到兩者相輔相成的概念



每一個都有自己的對應剖析方法

例如創見表格就會對應CreateTableData

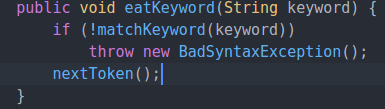
刪除資料對應DeleteData

查詢資料對應QueryData

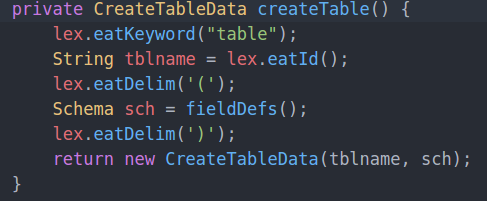
都有自己的剖析方式，由主程式Parser來呼叫他們

就好比C++裡面自己定義函數的模組化原理一樣，相當物件導向

剖析切割後，如果不符合文法便會報錯，BadSyntaxException();　例如eatKeyword吃到不符合規範的關鍵自就會報錯(細部則交由Parser進行)，如下圖所示

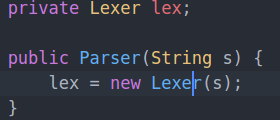


/Lexer



/Parser

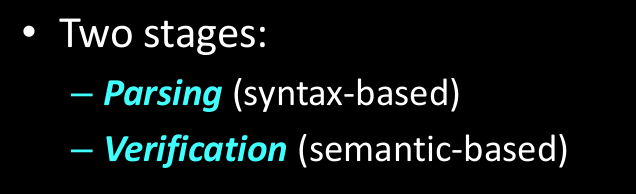
(lex代表呼叫Lexer又再一次呼應了Lexer&Parser兩者相輔相成)



/Parser

切割完關鍵字後，如果沒有BadSyntaxException，便會是一個沒有語法錯誤的SQL語句，此時

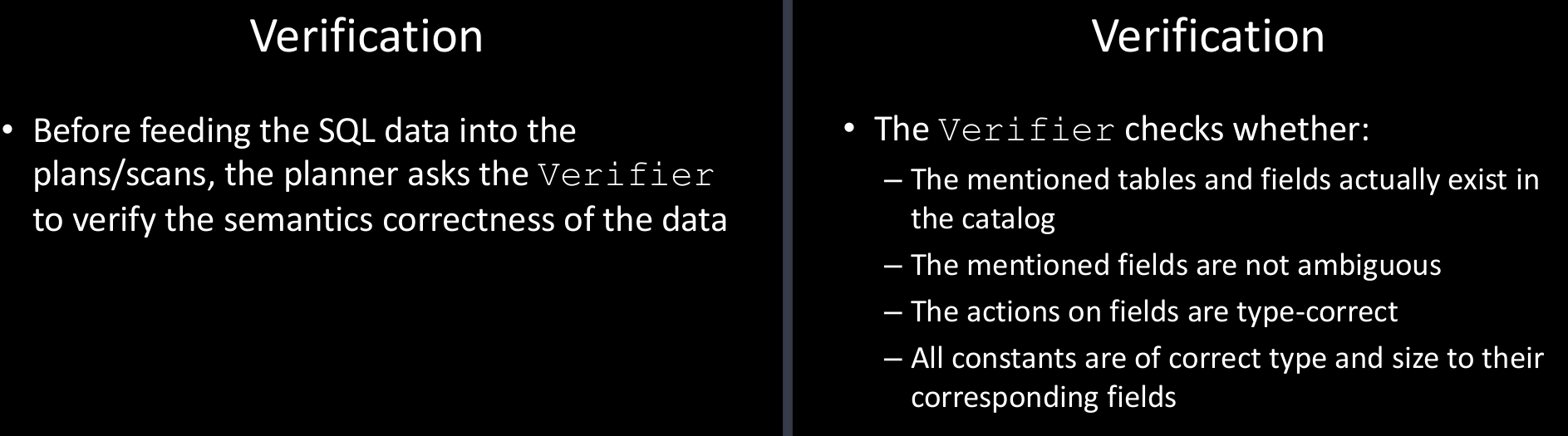
可以送到Verifier.java中執行，進行要真正做的事情。

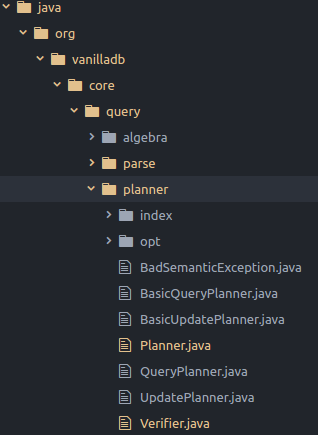


●step2語句邏輯分析，確認是一個合乎語意的SQL “Semantic Correctness”

切割剖析後當然就會把切割剖析好的東西送去分類以及查詢，在此已經準備進入

執行階段了，因此改由Planner資料夾中的Verifier負責語意正確的判斷(注意語法正確並不代表語意正確，一個是SyntaxCorrect，另個是SemanticCorrect兩者不同，方才Parsing procedure只有確認語法正確，也就是文法正確性，好比英文的時態或是主動詞搭配是否正確，現在是要看有沒有合乎邏輯的內容，合理的內容)





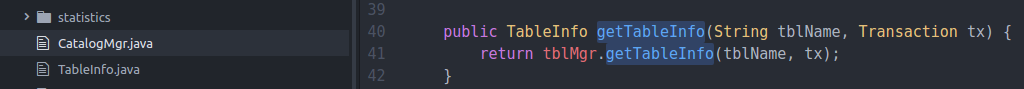
首先會創立兩個陣列分別存放要查詢的表格以及表格內要查什麼，在此可以看到使用List的資料結構



在此次作業我們分析Query data 所以用到verifyQueryData，首先檢查欲查詢表格是否存在



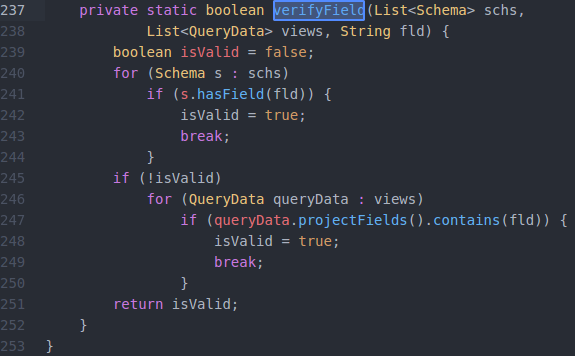
額外呼叫了位於core/strorage/metadata的CatalogMgr(類別管理元，建下面第二章圖片)中的getTableInfo來把table裡面的東西傳出來至Verifier比對，確認是否有那些表格，如果沒有就報錯說無表格(Table does not exist)(在此是沒有採用View虛擬表格的情況下)



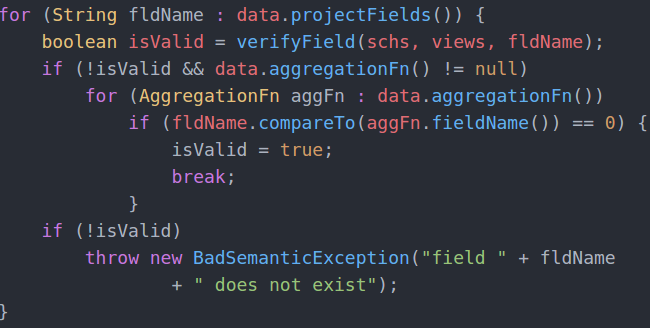
檢查表格後還要看表格的欄位，無論是一般查詢、聚合查詢(aggregation)、分組或是排序

都要檢查有無表格的該欄位才知道能否做事，由Verifier上半部提出各種認證，在交由下半部統一認證

（左為Verifier的上，右為下）



仔細來看可以發現每一個檢查欄位的邏輯都相當類似，皆會先有一個isValid來判斷是否存在，若不存在但是確有聚合函數(aggregationFn!=null)也依然可用(因為對於例：select XXX　YYY ZZZ aggFn(KKK)中剛剛只檢視了XXX但我們還沒檢視到聚合函數中的東西，即aggFn中的KKK因為只要他在表格中視一個有效的欄位亦能查詢，就把isValid設回真)



除非真的什麼都沒有才會丟出BadSemanticException表示邏輯內容無效的句子

其他的也是利用類似的方法檢定，只是Group by 和Sort by不需要聚合函數(正確而言是

SQL不支援這種寫法)

Semantic and Syntax均無誤後變可以開始進行Query了(以下Step3)

●step3 優化分析與代價查詢

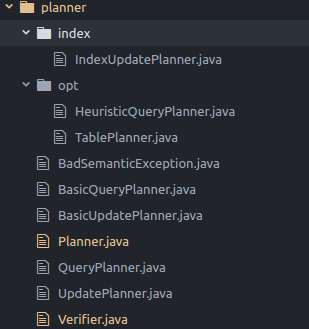
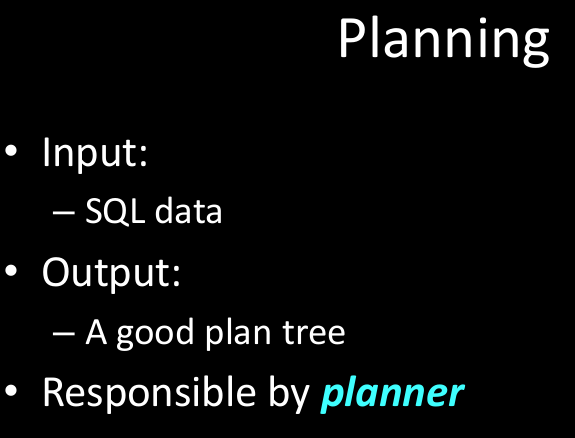
在Query開始前，回顧一下這次作業第二題的Query是

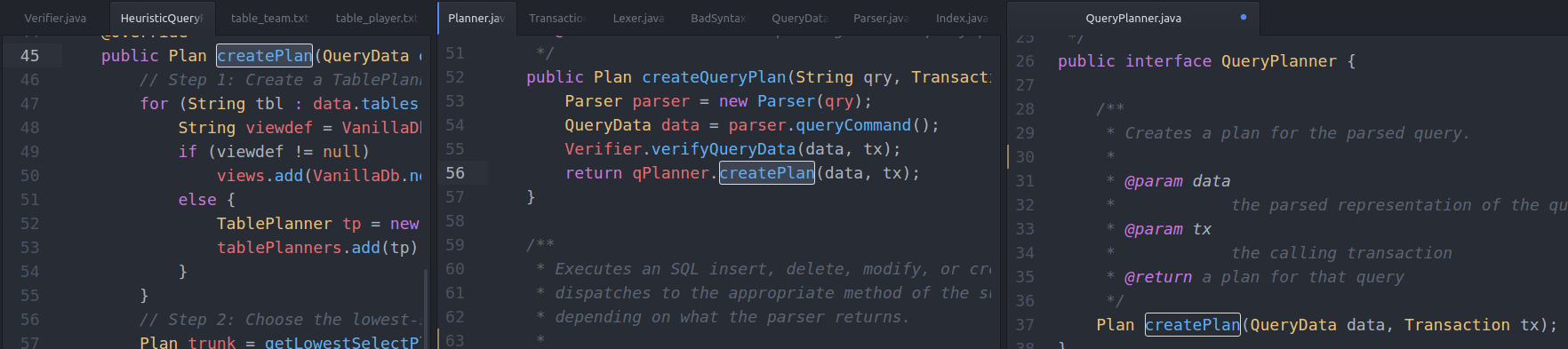
select team\_team\_id,count(player\_player\_id) from team,player where

team\_team\_id=player\_team\_id and category='badminton' group by

team\_team\_id

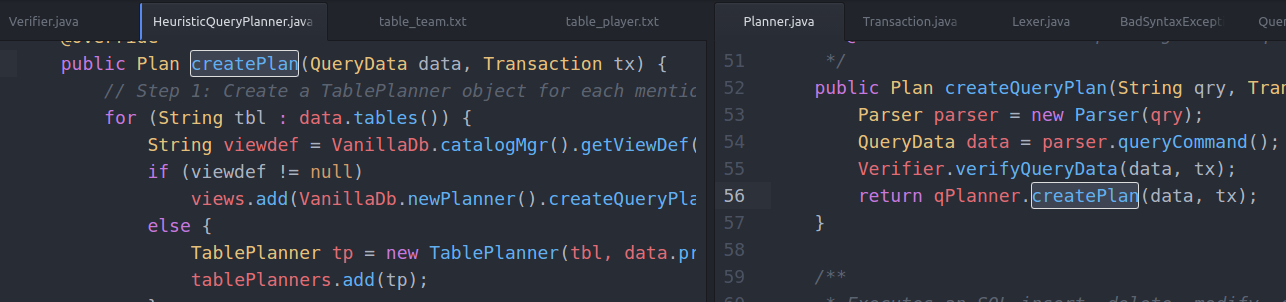
在正式查詢前會先優化Query ，由Planner負責，看看那一個查訊計畫樹比較優



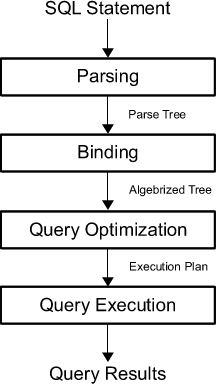
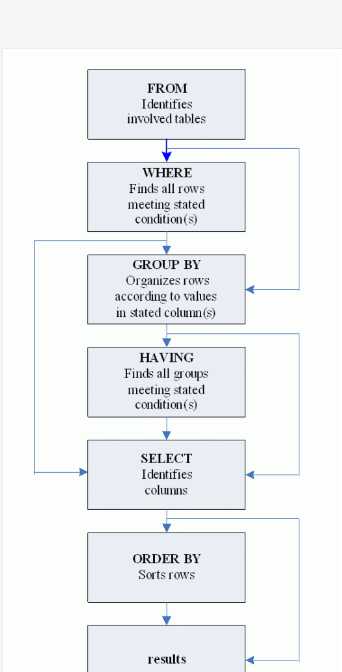
HeuristicQueryPlanner(啟發式查詢規劃器，以下簡稱HQP)便是整個Planner的核心。例如這次要做Query 則QueryPlanner將會呼叫Planner(下圖中間)接著Planner再呼叫HQP

HQP(上圖左邊)，Planner會把剖析好的資料(即要查詢的表格與欄位等等）送到HQP中來

createPlan，如下圖所示。

---------------------------------------以下為HQP詳細流程/程式碼分析------------------------------------

以下是SQL執行的流程

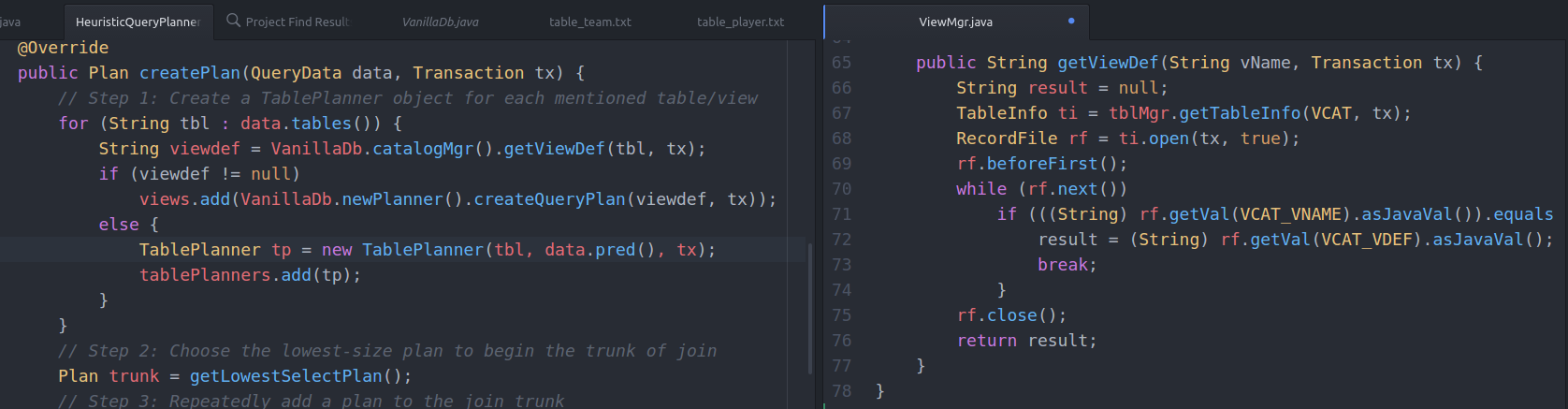


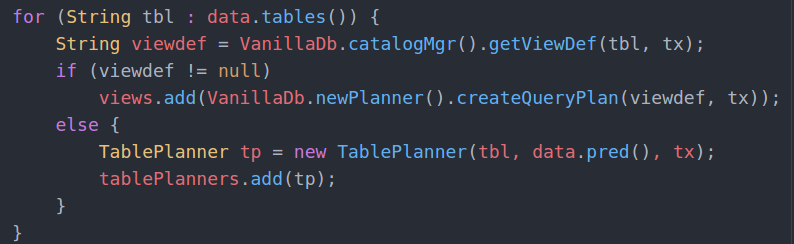
左為一個Query子句查詢的順序右為整個SQL從輸入到結果出來的順序，可見，我們現在是在Query Optimization和Query Execution之間，欲尋找一個代價最低的計畫。

由上左圖可以看出，在一個SQL執行語句中首先可以看出，會先找出所要查詢的表格有那些，因此第一個進到的就是方才所述的『表格計畫』create a table planner如下圖所示

會呼叫ViewMgr來把表格的欄位以輸入進去的查詢句檢視getViewDef(tbl,tx)

接著若查詢為真if (viewdef != null)，代表有查詢到該表格，則新增該查詢計畫，views.add(VanillaDb.newPlanner().createQueryPlan(viewdef, tx));



第二步驟，會檢視接下來要Select的資料，雖然和方才提到的SQL執行順序不同(那張圖片的SQL select並不是在第二步驟)，但透過先把Select確認好可以在Join或是Cartesian product等等未來的表格互動動作省下一些不必要的比較，如下圖所示。



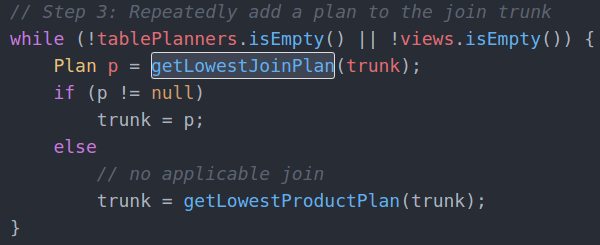
以下第二張圖為getLowestSelectPlan詳細的演算法，額外呼叫core/query/algebra/SelectPlan.java要資料來計算每一種selectPlan的優化程度，並且由recordsOutput實際去計算

逐步更新出最低cost的plan(※因為沒有使用View所以暫時不討論)，由以下第一張圖可以看出在SelectPlan.java詳細的討論各種數據出現的頻率來計算出每一個select的cost





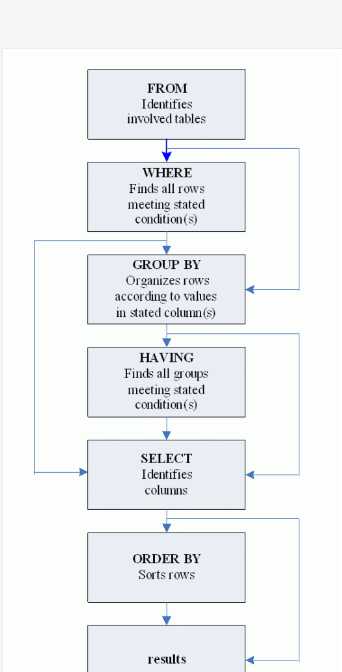
把欲查詢的欄位最佳化、去除不必要的欄位後，接著對表格們檢查是否要Join，否則是一般的Cartesian product(均要找最低代價組合，所以跟剛剛一樣的概念，再跑一次代價查詢與比較的邏輯，只是這次換成Join的版本)，如以下三張圖片的程式碼邏輯，如果recordsOutput就選擇它，因為代表刪選資料篩的更詳細，減少不必要的查詢。

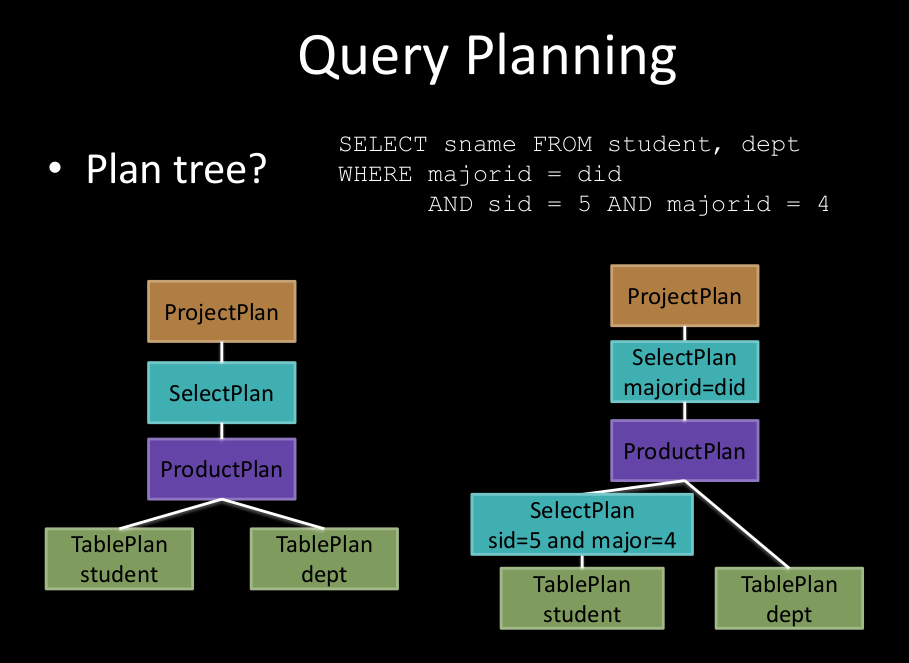






最後分析分組計畫(Groupby Plan)和投射計畫(Project Plan)，剛好也是第二題的最後幾步驟

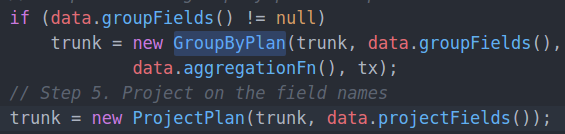
找最低代價的邏輯也是跟方才類似，但大家一定會想說方才不是已經有找過select了，為什麼現在又要在找一次？

原因如下圖QueryPlanning(請原諒Linux LibreOffice凌亂的排版)

方才第一個Select Plan其實是『前置篩選』，去除掉大部無用資料，例如上圖表格中右邊的查詢效能一定會比左邊好，因為在笛卡兒積之前如果把無用的資料去除將可以省下相當的代價（以下解釋右上圖中例子）

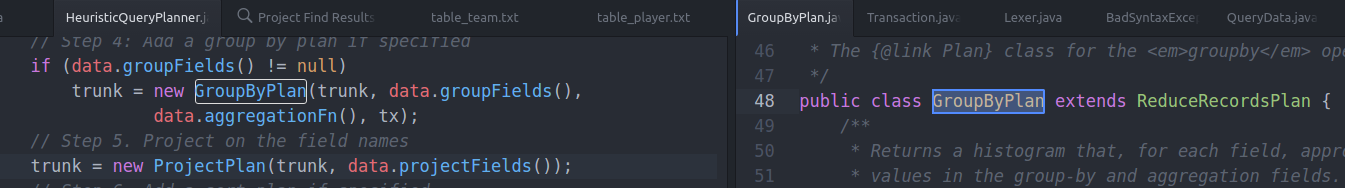
假設學生有一萬人、科系有一百個，則CartesianProduct會花費O(student\*dept)的查詢時間複雜度，把必要的列資料抓出來（在此極為sid=5 and major=4）、可能只有數位學生符合條件甚至一個，因此在CartesianProduct前已經可以把student的數量大大降低，雖然還是有點Bruteforce的味道，卻是相當有智慧的Bruteforce（暴力解）法則。

而在此的第二個Select Plan(即下圖的ProjectPlan)



就是看有哪些欄位要被提出罷了，實際上此Plan並無計畫的味道，因為方才（上一段笛卡兒積代價評估）已經做過了。

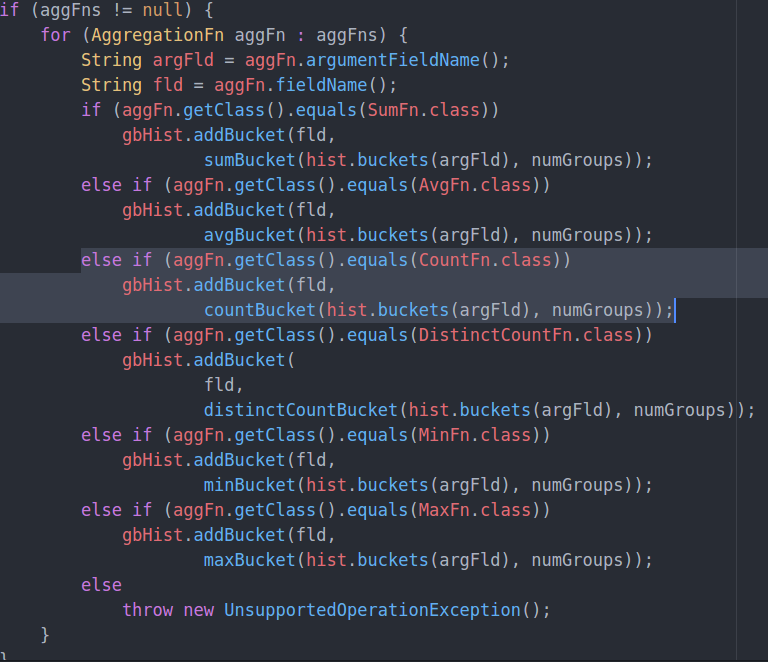
先前還有一個分組評估邏輯，評估方法和之前類似，都會呼叫固定專門處理的java檔案來計算出代價，如下圖的結構，。

由函數的命名法則可以發現，分組事實上並不太需要研究何者的代價最低，因為分組就只是單純的把資料分群化而已，真正會影響時間複雜度的是在之前做Cartesian product的時候，如果有預先篩選資料則可以大大的降低時間複雜度，因此在Group by這裡只有trunk = new GroupByPlan(trunk, data.groupFields(),

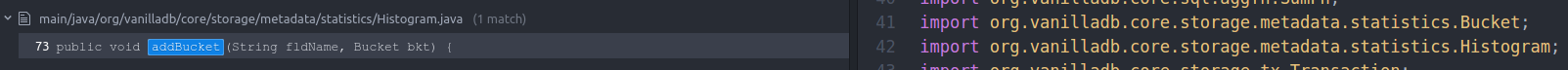
data.aggregationFn(), tx);

就單純的直接執行」Group by的代價計算而無之前的擇優演算法，例如getLowestSelectPlan等等。

第二題的Query有用到aggregation function ，也就是count來計算說打羽球的球隊中，每一隊有多少人，最後依據羽球的球隊編號來統計該隊伍有多少人。

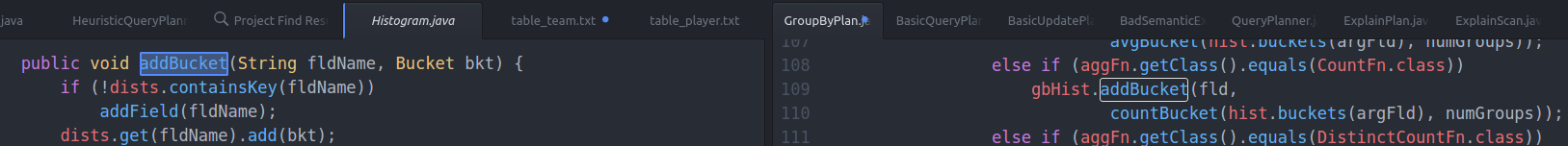
既然是count函數那當然會有對應的count來確認說該分組原則下每一組有多少資料，所以在此呼叫了countBucket，以及addBucket。

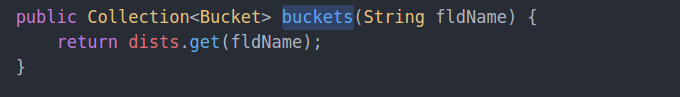
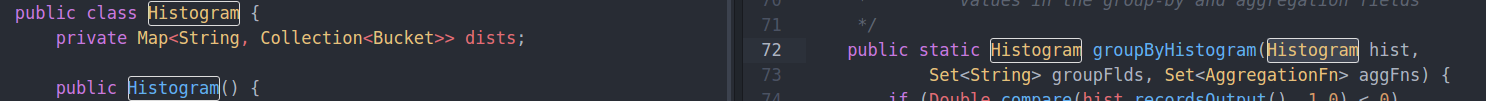
詳細流程如所述：首先對於每一個出現的分組元素，亦即每一個『組』會先把這個組加進bucket，在此GoupByPlan呼叫了Histogram，如下圖所示，很明顯放在



org.vanilladb.core.storage.metadata.statistics.Histogram，由命名可見是一個統計數據用的功能，把模組化切割、各司其職又在一次呼應了java物件導向程式開發的觀念。

之後呼叫了addBucket，來將分組後的每一『組』創造出來，如下圖所示，由GroupByPlan.java來呼叫Histogram.java 中addBucket

接著便Count這個組別的內容物，呼叫了countBucket / countBucket(hist.buckets(argFld), numGroups));



/Histogram.java

由函數的定義可看見hist.buckets(argFld)可以把對於某個filedName作為分群的法則，在此一個群就是一個bucket(桶)，所以最後要看每個桶中有多少個資料就呼叫sumBucket，其中它會回傳一個return new Bucket(sumRange, numGroups, numGroups);代表一個已經依據分群原則分好的桶，亦即，Group。

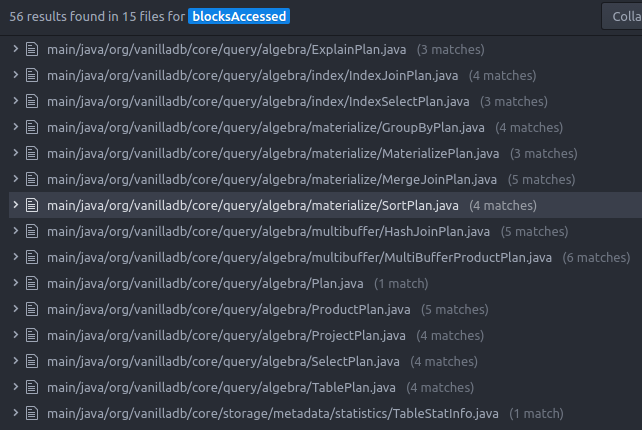
其中較上方的for (Bucket bkt : dist)註解　// estimate sumLow as the only one smallest value in a group　代表把該群中最小值抓出

另外下方的　for (Constant high : desc)註解　// estimate sumHigh as the sum of top maxGroupSize values　代表把該群中最大值抓出

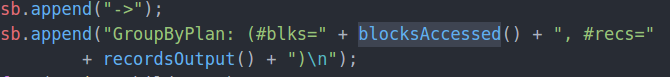
最後透過ConstantRange sumRange = ConstantRange.newInstance(sumLow, true,sumHigh, true);　來確認一個群體的範圍同時也求出該群中欲加總的元素個數。

●step4 Explain SQL Query以及優化後的抉擇結果

對於每一個SQL Relational Algebra都會分析該代價，查詢了多少區塊以及最後輸出多少條結果，如下圖所示的blockAccessed(區塊查詢)

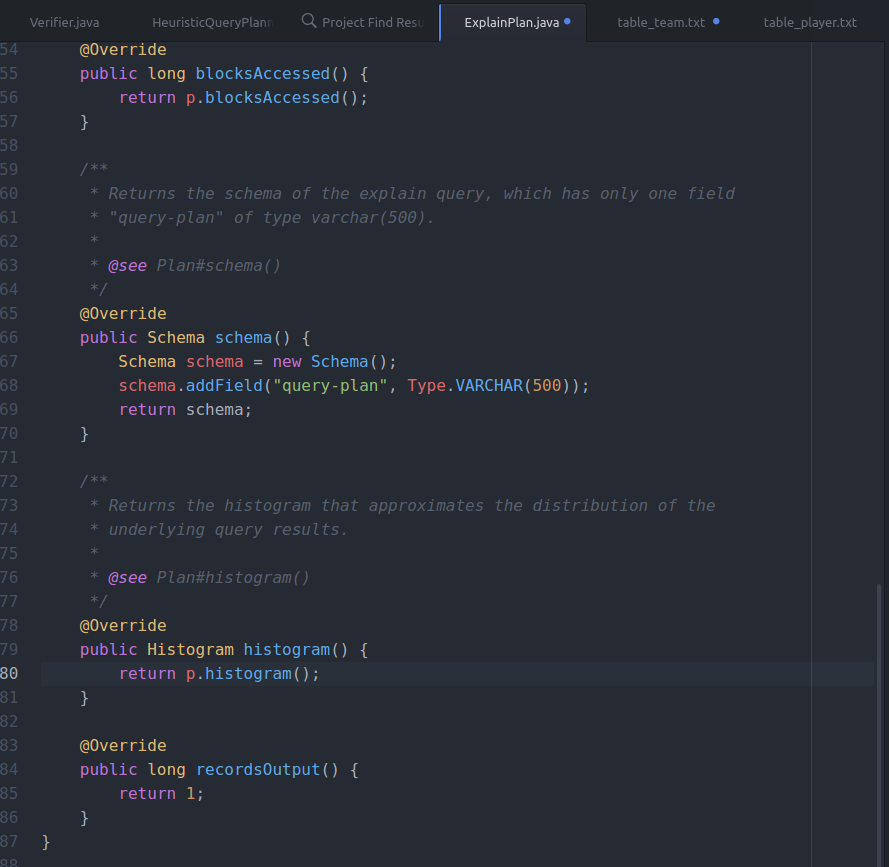


會在explain plan的時候被顯示出來，例如GroupByPlan中的

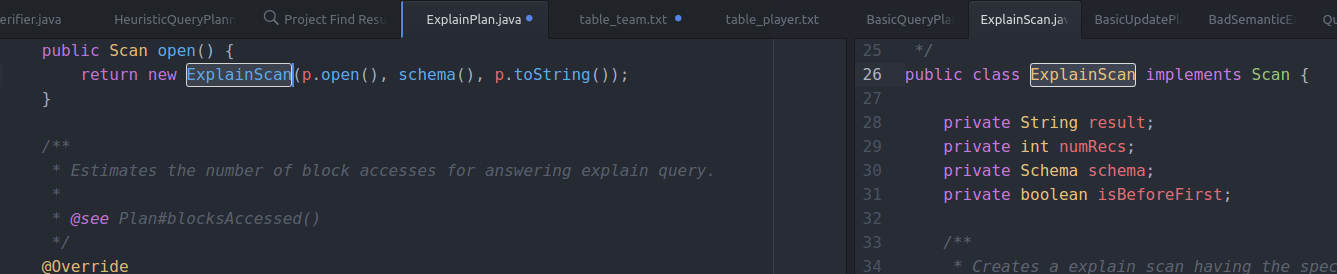


欲explain整個SQL的查詢時，首先會進到　ExplainPlan，整體最終的輸出資料會在ExplainScan.java中被輸出，把每一次運用到 Relational Algebra所涉及的資料數量都count

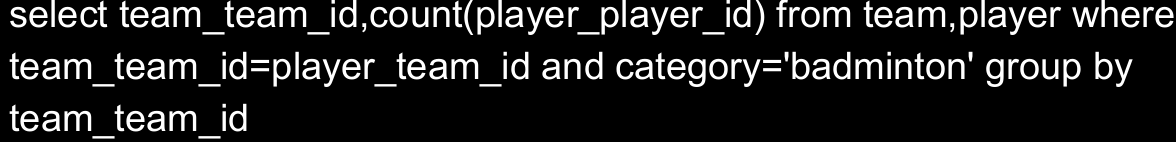
(各式各樣的blockAccessed 有product、join、group等等）



並輸出至ExplainScan顯示出來，ExplainPlan和ExplainScan兩者相輔相成，前者提出計畫的結果後者進行計畫內容的掃描，名符其實，如下圖所示的呼叫情形與檔名架構。

所有完成後便是一個完整分析的查詢

此次的SQL Query

以及資料的內容