**武汉大学计算机学院**

**本科生实验报告**

**数据库系统实现**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：数据库系统实现

团 队 名 称 ：九转大码品鉴队

指 导 教 师 ：

团 队 成 员 一：

团 队 成 员 二：

团 队 成 员 三：

团 队 成 员 四：

团 队 成 员 五：

团 队 成 员 六：

二○二三年五月

**郑 重 声 明**

本团队呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

团队成员签名： 日期：

摘 要

本实验的实验目的是基于武汉大学自研的TOTEM数据库和高德地图API在安卓操作系统上实现记录运动轨迹数据并可视化，从而加深学生对数据库系统实现层面的理解以及动手能力。

本实验使用Windows 10作为开发环境，Android Studio作为开发工具，JAVA作为开发语言，Android操作系统作为部署平台。

实验的主要内容包括三个任务。任务一要求设计系统表，实现系统表的读取和保存；任务二要求实现SQL解析中的对象JOIN操作；任务三要求实现轨迹数据的JOIN和可视化。

本实验最终实现了在Android应用上实现用户的轨迹跟踪和可视化，以及轨迹的连接操作。

本实验的源代码发布在**<https://github.com/cylqqqcyl/whu-tmdb>**，将在实验周期结束后公开。

**关键词：**TOTEMDB；数据库系统实现；JOIN

目 录

[摘 要 3](#_Toc3896)

[1 实验目的和意义 6](#_Toc23118)

[1.1 实验目的 6](#_Toc17743)

[1.2实验意义 6](#_Toc5485)

[2 实验设计 7](#_Toc20447)

[2.1 概述 7](#_Toc17818)

[2.2 实验原理 7](#_Toc17641)

[2.2.1 TOTEM数据库 7](#_Toc30317)

[2.2.2 Android Studio 7](#_Toc8380)

[2.2.3 高德地图API 8](#_Toc15787)

[2.2.3 JOIN操作 8](#_Toc18933)

[2.3 实验方案 8](#_Toc6086)

[2.3.1 设计系统表 9](#_Toc17018)

[2.3.2 实现SQL解析中的对象JOIN操作 15](#_Toc341)

[2.3.3 实现轨迹数据的JOIN和可视化 17](#_Toc21901)

[2.3.4 测试与演示 22](#_Toc32124)

[结论 23](#_Toc21278)

[参考文献 24](#_Toc30805)

**1 实验目的和意义**

**1.1** **实验目的**

本实验的目的是通过实践操作，让学生深入理解数据库系统的基本概念和原理，掌握数据库系统的设计、实现、管理和维护技术，培养学生的数据库应用开发能力和解决实际问题的能力。同时，通过实验，学生还能了解数据库系统的应用领域和未来发展趋势。

**1.2实验意义**

该实验是理论知识和动手能力的综合体现。通过本实验，学生可以更加深入地理解数据库系统的基本概念和原理和培养实际操作能力。同时，使用github平台多人协同开发有助于培养学生的团队协作能力。

**2 实验设计**

**2.1 概述**

本实验实现了在Android操作系统上基于TOTEM数据库和高德地图API记录并可视化用户行动轨迹的应用功能开发工具为Android Studio，编程语言为JAVA，开发环境为Windows 10操作系统，应用的部署环境为Android操作系统。

实验步骤主要包括：

1. 设计系统表，实现系统表的读取和保存；
2. 实现SQL解析中的对象JOIN操作；
3. 实现轨迹数据的JOIN和可视化。

**2.2 实验原理**

**2.2.1 TOTEM数据库**

为了管理结构复杂、语义丰富的大数据，武汉大学珞珈图腾数据库实验室在国家863数据库重大专项课题支持下研制了对象代理数据库系统TOTEM[1]。TOTEM数据库基于我们提出的具有原创性的对象代理模型。该模型既具有关系数据模型的柔软性又具有面向对象数据模型表现复杂语义能力，发表在了国际数据库顶级会议IEEE International Conference on Data Engineering 和权威期刊 IEEE Transaction on Knowledge and Data Engineering 上得到学术界认可和引用。相比目前世界上普遍采用的对象关系数据库管理系统，对象代理数据库系统能够提供灵活对象视图、动态分类、跨类查询等先进功能，形成了自主知识产权，获教育部高等学校科技进步二等奖。它是大数据与人工智能时代的应需之作，可广泛应用于非结构、跨媒体、多模态数据管理与分析。

**2.2.2 Android Studio**

Android Studio[2][3]是谷歌Android操作系统的官方集成开发环境（以下简称IDE），基于JetBrains的IntelliJ IDEA软件开发，并专门为Android开发而设计。它可以在Windows、MacOS和Linux操作系统上下载使用，是Eclipse Android开发工具（E-ADT）的替代品，成为本地Android应用程序开发的主要IDE。

Android Studio于2013年5月16日在Google I/O大会上宣布，并从2013年5月的版本0.1开始进入早期预览阶段，随后从2014年6月的版本0.8开始进入Beta阶段。第一个稳定版本从2014年12月的版本1.0开始发布。到2015年底，谷歌停止支持Eclipse ADT，使得Android Studio成为Android开发的唯一官方支持的IDE。

**2.2.3 高德地图API**

高德地图Android SDK[4]是一套地图开发调用接口，开发者可以轻松地在自己的Android应用中加入地图相关的功能，包括：地图显示（含室内、室外地图）、与地图交互、在地图上绘制、兴趣点搜索、地理编码、离线地图等功能。

高德地图Android SDK 专业版是在Android SDK已有服务的基础上，新增支持了自定义地图在线加载、自定义地图元素纹理等功能，便于开发者完成基于自身场景的更深层、更个性化地图的开发需求。

**2.2.3 JOIN操作**

在SQL中，JOIN语句将一个或多个表的列组合成一个新表。该操作对应于关系代数中的联接操作，即联接将两个表拼接在一起，并将具有匹配字段的记录放在同一行上。JOIN操作包括 INNER、LEFT OUTER、RIGHT OUTER、FULL OUTER和CROSS。

本实验的应用场景是自动驾驶行车轨迹数据和手机轨迹数据用了两张表分别存储，导致了大量的数据冗余。通过对两张表施加JOIN操作，可以消除大量重复数据，便于对数据进行分析和挖掘以及提高数据的准确性和可靠性。

**2.3 实验方案**

本实验主要包括设计系统表、 实现SQL解析中的对象JOIN操作和实现轨迹数据的join和可视化三个任务。

**2.3.1 设计系统表**

系统表（system table）是存储数据库系统元数据（metadata）的表。元数据是描述数据库中各种对象的数据，如表、列、索引、视图、存储过程等。系统表存储这些元数据信息，包括对象的名称、类型、所有者、创建时间、修改时间、大小等信息。其作用包括支持数据库系统的运行和管理，以及提供元数据信息供开发人员和系统管理员使用。

本实验中主要用到了ObjectTable，SwitchingTable，DeputyTable，BiPointerTable，ClassTable共五张系统表。

在本实验代码中，上述系统表相关的代码基本完善。为了提高开发效率，本组开发人员进一步在DeputyTable中追加了deputyname字段并正确让其显示在Android应用上。

首先是向DeputyTable中添加字段以及修改相关函数。在DeputyTableItem.java文件中给出了DeputyTableItem类的实现。在原先的基础上，为类添加字符串类型的deputyname字段，表示deputy所属的类。同时，还需修改类中的函数equals()和hashCode()。

1. **public** class DeputyTableItem implements **Serializable** {
2. **public** DeputyTableItem(**int** originid, **int** deputyid, String deputyname,String[] deputyrule) {
3. this.originid = originid;
4. this.deputyid = deputyid;
5. this.deputyrule = deputyrule;
6. this.deputyname = deputyname;
7. }
9. **public** DeputyTableItem() {
10. }
12. **public** **int** originid = 0;            //类id
13. **public** **int** deputyid = 0;           //代理类id
14. **public** String deputyname = "";
15. **public** String[] deputyrule = new String[0];    //代理guizedui

18. @Override
19. **public** boolean equals(Object object){
20. if(this==object) **return** **true**;
21. if (!(object instanceof DeputyTableItem)) {
22. **return** **false**;
23. }
24. DeputyTableItem oi=(DeputyTableItem) object;
25. if(this.originid!=oi.originid){
26. **return** **false**;
27. }
28. if(this.deputyid!=oi.deputyid){
29. **return** **false**;
30. }
31. if(this.deputyrule!=oi.deputyrule){
32. **return** **false**;
33. }
35. **return** Objects.equals(this.deputyname, oi.deputyname);
36. }
38. @Override
39. **public** **int** hashCode() {
40. **int** result = 17;
41. result = 31 \* result + Objects.hash(this.originid)+Objects.hash(Arrays.stream(this.deputyrule).toArray())+Objects.hash(this.deputyid)+Objects.hash(this.deputyname);
42. **return** result;
43. }
44. }

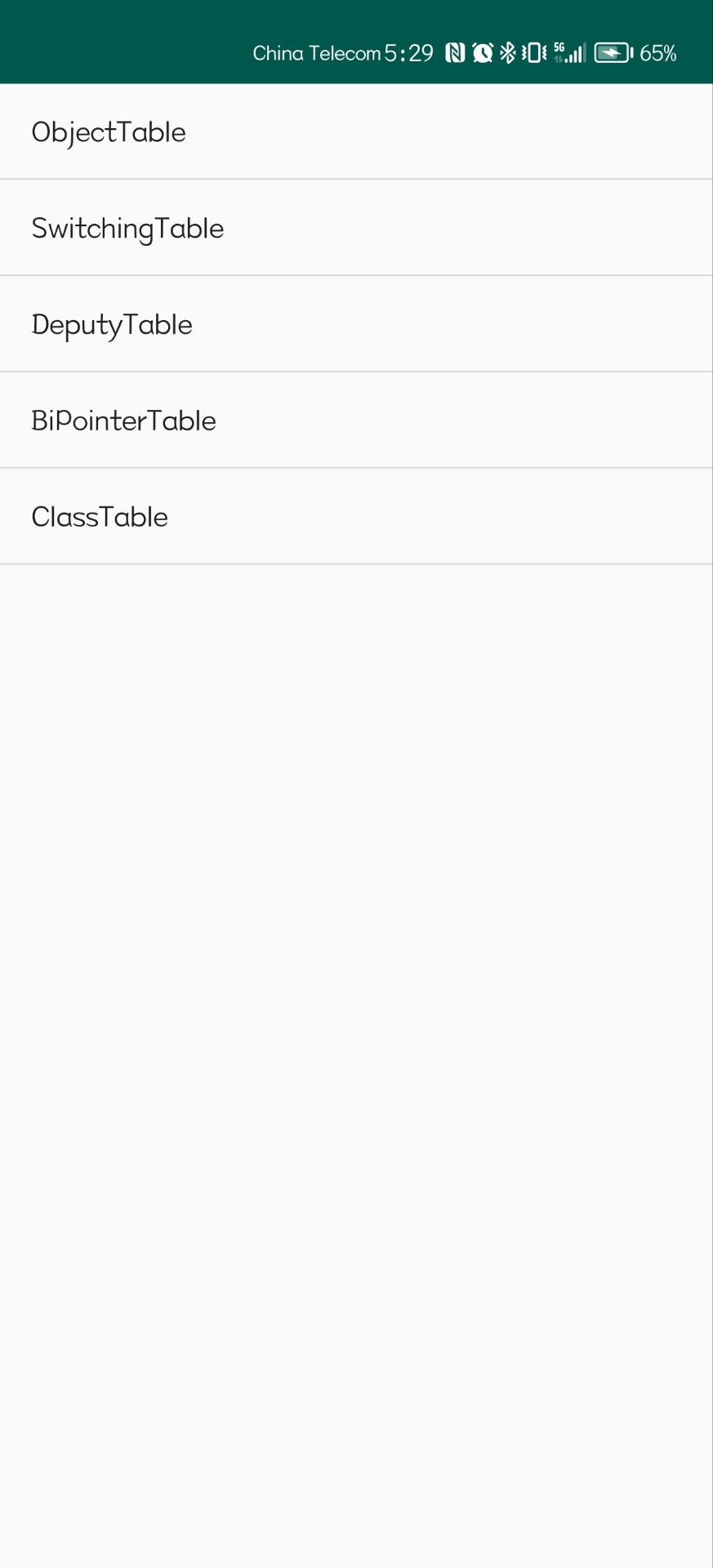
然后是修改向DeputyTable中插入元组的相关函数。在CreateDeputyClassImpl.java文件中给出了insertDeputyTable()函数的实现。为了插入deputyname，为其添加参数deputyClass。

1. **public** void insertDeputyTable(String[] className,**int** deputyType, **int** deputyId, String deputyname) throws TMDBException {
2. HashSet<String> collect = Arrays.stream(className).collect(Collectors.toCollection(HashSet::new));
3. **for** (String s :
4. collect) {
5. **int** oriId=memConnect.getClassId(s);
6. memConnect.getDeputyt().deputyTable.**add**(
7. new DeputyTableItem(oriId,deputyId,deputyname,new String[]{deputyType+""})
8. );
9. }
10. }

最后是让该字段正确显示在应用上。在ShowDep.java文件中给出了ShowDep类的实现，用于显示deputyTable。修改tabCol的值并添加stemp4变量来表示deputyname。然后将deputyTable的各个属性依次显示。

1. **public** class ShowDep extends AppCompatActivity implements **Serializable** {
2. private final **int** W = ViewGroup.LayoutParams.WRAP\_CONTENT;
3. private final **int** M = ViewGroup.LayoutParams.MATCH\_PARENT;
4. private TableLayout show\_tab;
5. //private ArrayList<String> objects = new ArrayList<String> ();
7. @Override
8. protected void onCreate(Bundle savedInstanceState) {
9. Log.d("ShowDep", "oncreate");
11. super.onCreate(savedInstanceState);
12. setContentView(R.layout.print\_result);
14. Intent intent = getIntent();
15. Bundle bundle0 = intent.getExtras();
16. showDepTab((DeputyTable) bundle0.getSerializable("DeputyTable"));
18. }
20. private void showDepTab(DeputyTable deputyt) {
21. **int** tabCol = 4;
22. **int** tabH = deputyt.deputyTable.**size**();
23. Object oj1,oj2;
24. String stemp1, stemp2, stemp3, stemp4;
25. String[] satemp;
27. show\_tab = findViewById(R.id.rst\_tab);
29. **for** (**int** i = 0; i <= tabH; i++) {
30. TableRow tableRow = new TableRow(this);
31. if (i == 0) {
32. stemp1 = "originid";
33. stemp2 = "deputyid";
34. stemp3 = "deputyrule";
35. stemp4 = "deputyname";
36. } **else** {
37. oj1 = deputyt.deputyTable.get(i-1).originid;
38. oj2 = deputyt.deputyTable.get(i-1).deputyid;
39. satemp = deputyt.deputyTable.get(i-1).deputyrule;
40. stemp1 = oj1.toString();
41. stemp2 = oj2.toString();
42. stemp3 = satemp[0].toString();
43. stemp4 = deputyt.deputyTable.get(i-1).deputyname;
44. }
45. **for** (**int** j = 0; j < tabCol; j++) {
46. TextView tv = new TextView(this);
47. switch (j) {
48. case 0:
49. tv.setText(stemp1);
50. break;
51. case 1:
52. tv.setText(stemp2);
53. break;
54. case 2:
55. tv.setText(stemp3);
56. break;
57. case 3:
58. tv.setText(stemp4);
59. break;
60. }
61. tv.setGravity(Gravity.CENTER);
62. tv.setBackgroundResource(R.drawable.tab\_bg);
63. tv.setTextSize(28);
64. tableRow.addView(tv);
65. }
66. show\_tab.addView(tableRow, new TableLayout.LayoutParams(M, W));
68. }
70. }
72. }

最终系统表在应用上显示的效果如下图所示。





**2.3.2 实现SQL解析中的对象JOIN操作**

针对本实验的应用场景，基于已有的SELECT语句可以实现与JOIN等价的功能。

主要思想是先将多个表做笛卡尔积，然后在需要连接的属性上用WHERE语句判断是否相等已选出目标元组。例如，要将table1和table2分别在column1和column2上连接。SQL语句的实现代码如下所示。

1. **SELECT** \* **FROM** table1, table2 **WHERE** table1.**column** = table2.**column**;

本实验代码中，在Tuple.java文件中给出了元组的定义。每个元组的包含了三个属性，即trajectory\_id、user\_id和trajectory。其中，trajectory是一个字符串，表示相邻两点的经纬度信息，格式为“经度1-维度1-经度2-维度2”。

在TJoinSelect.java文件中给出了TJoinSelect类的实现，关键代码如下图。

1. **public** **class** TJoinSelect extends SelectImpl{

4. **private** MemConnect memConnect;
6. **public** TJoinSelect(MemConnect memConnect) {
7. super(memConnect);
8. **this**.memConnect = memConnect;
9. }
11. **public** TJoinSelect() {
13. }
15. //TODO TMDB
16. //重写select的intersect方法，使其使用trajectory similarity join 进行连接
17. @Override
18. **public** SelectResult intersect(SelectResult left, SelectResult right){
19. LongestCommonSubSequence longestCommonSubSequence=**new** LongestCommonSubSequence();
20. //新建tuplelist，存储两表intersect之后的结果
21. TupleList res=**new** TupleList();
22. //遍历左表的tuple
23. **for** (**int** i = 0; i < left.getTpl().tuplelist.size(); i++) {
24. //获取当前tuple
26. //调用TrajTrans的getTraj方法，将tuple中的String轨迹转换成List<Coordinate>的形式，得到traj1
27. Tuple tuple = left.getTpl().tuplelist.get(i);
28. List<Coordinate> leftTraj = TrajTrans.getTraj((String) tuple.tuple[2]);
29. //遍历右表的每个tuple
30. **for** (**int** j = 0; j < right.getTpl().tuplelist.size(); j++) {
31. //获取当前右表tuple
32. //并通过TrajTrans的getTraj方法得到List<Coordinate>，得到traj2
33. Tuple rightTuple = right.getTpl().tuplelist.get(j);
34. List<Coordinate> rightTraj = TrajTrans.getTraj((String) rightTuple.tuple[2]);
35. //通过longestCommonSubSequence的getCommonSubsequence方法得到traj1和traj2的公共子序列，theta值自设
36. List<Coordinate> commonSubsequence = longestCommonSubSequence.getCommonSubsequence(leftTraj, rightTraj, 3);
37. //通过得到的子序列的长度设置阈值，判定当前子序列是否值得加入结果集合中
38. **if**(commonSubsequence.size()>=1){
39. //如果满足，则新建加入到结果结合中的tuple
40. //此tuple其它的部分与左表的当前tuple全部一致，除了轨迹段改为公共子序列
41. Tuple temp=**new** Tuple();
42. temp.tupleId=tuple.tupleId;
43. temp.tupleIds=tuple.tupleIds;
44. temp.tuple=tuple.tuple;
45. //需要将得到的轨迹子序列，转换成string的形式，然后将tuple中轨迹部分设置为转换后的值
46. String temps=TrajTrans.getString(commonSubsequence);
47. temp.tuple[2]=temps;
48. //在新建的tuplelist中加入当前tuple
49. res.tuplelist.add(temp);
50. **break**;
51. }
52. }
53. }
54. //将左表的selectResult 也就是left的tuplelist设置为新的结果集
55. left.setTpl(res);
56. //返回新的selectrResult
57. **return** left;
58. }
59. }

通过重写intersect函数可以实现trajectory之间的连接操作。遍历左表和右表，对于左表中的元组leftTuple和右表中的元组rightTuple，首先使用TrajTrans.getTraj()方法将元组中的trajectory转换成坐标格式；其次调用getCommonSubsequence()方法获取两个trajectory的最长公共子串。其作用是衡量两个trajectory之间的相似度。对于最长公共字串长度达到阈值，即轨迹trajectory足够相似的两个元组，将其加入到连接的结果之中。遍历完左表和右表后就可以得到最终的连接结果。

一个使用该连接方法的SQL语句如下所示。该语句将watch\_traj表和mobile\_traj通过intersect的方式连接成tj表。

1. **create** tjoindeputy tj **as** **select** \* **from** watch\_traj **intersect** **select** \* **from** mobile\_phone\_traj;

**2.3.3 实现轨迹数据的JOIN和可视化**

本实验中，当用户移动时，应用需要实时记录并绘制出其行动轨迹。本组开发者实现了实时（real-time）和离线（offline）两种可视化轨迹的方式。

在MapActivity.java文件中给出了onLocationChanged()函数的实现。该函数在捕捉到用户位置变动时记录其经纬度、时间和用户ID等信息，并将新位置的经纬度加入到trace数组中。之后使用PolylineOptions的addAll()方法把trace中的点加入到线段坐标点集合并指定颜色，得到options。该方式支持实时轨迹跟踪和可视化。

1. **public** **void** onLocationChanged(AMapLocation amapLocation) {
2. **if** (mListener != **null** && amapLocation != **null**) {
3. **if** (amapLocation != **null**
4. && amapLocation.getErrorCode() == 0) {
5. mLocationErrText.setVisibility(View.GONE);
6. LatLng location = **new** LatLng(amapLocation.getLatitude(), amapLocation.getLongitude());
7. **if** (!mFirstFix) {
8. mFirstFix = **true**;
9. addCircle(location, amapLocation.getAccuracy());//添加定位精度圆
10. addMarker(location);//添加定位图标
11. mSensorHelper.setCurrentMarker(mLocMarker);//定位图标旋转
12. aMap.moveCamera(CameraUpdateFactory.newLatLngZoom(location, 18));
13. } **else** {
14. mCircle.setCenter(location);
15. mCircle.setRadius(amapLocation.getAccuracy());
16. mLocMarker.setPosition(location);
17. }


21. **double** latitude = amapLocation.getLatitude(); // 获取纬度
22. **double** longitude = amapLocation.getLongitude(); // 获取经度
23. Date date = **new** Date(amapLocation.getTime()); // 获取定位时间
24. String userID = Secure.getString(getContentResolver(), Secure.ANDROID\_ID); // 获取uid

27. List<LatLng> trace = **new** ArrayList<LatLng>();
28. **int** szT = trajectory.size();
29. **if**(szT>0) trace.add(**new** LatLng(trajectory.get(szT-1).latitude,trajectory.get(szT-1).longitude));
30. // 将定位点加入轨迹集合
31. trajectory.add(**new** TrajectoryPoint(longitude, latitude, date, userID));
32. trace.add(**new** LatLng(trajectory.get(szT).latitude,trajectory.get(szT).longitude));
33. PolylineOptions options = **new** PolylineOptions().addAll(trace).width(2).color(Color.argb(255, 0, 0, 255));
34. **if**(szT>0){
35. polyline = aMap.addPolyline(options);
36. **int** a=1;//cut
37. }
38. } **else** {
39. String errText = "定位失败," + amapLocation.getErrorCode()+ ": " + amapLocation.getErrorInfo();
40. Log.e("AmapErr",errText);
41. mLocationErrText.setVisibility(View.VISIBLE);
42. mLocationErrText.setText(errText);
43. }
44. }
45. }

对于离线模式，在MapActivity.java中给出了drawTrace()函数的实现。该函数读取历史轨迹。此外，对于每一条轨迹，使用不同的bitmap为其分配不同的轨迹颜色。

1. **private** **void** drawTrace(){
2. // 读取历史轨迹数据
3. ArrayList<ArrayList<TrajectoryPoint>> trajectories = TrajectoryUtils.load();
4. **if**(trajectories == **null** || trajectories.size() == 0)
5. **return**;
6. **int** counter = -1;
7. **for**(ArrayList<TrajectoryPoint> trajectory : trajectories){
8. counter = (counter + 1) % 11;
9. List<LatLng> trace = **new** ArrayList<LatLng>();
10. // 绘制每条轨迹
11. **for**(TrajectoryPoint point : trajectory){
12. // 绘制每个点
13. LatLng latLng = **new** LatLng(point.latitude, point.longitude);
14. aMap.addMarker(**new** MarkerOptions().position(latLng).icon(BitmapDescriptorFactory.fromResource(colorIcon[counter])));
15. trace.add(latLng);
16. }
17. Bitmap newBm = BitmapDescriptorFactory.fromResource(colorIcon[counter]).getBitmap();
18. PolylineOptions options = **null**;
19. **if** (android.os.Build.VERSION.SDK\_INT >= android.os.Build.VERSION\_CODES.Q) {
20. options = **new** PolylineOptions().addAll(trace).width(8)
21. .color(newBm.getColor(newBm.getWidth()/2,newBm.getHeight()/2).toArgb());
22. }
23. **else**{
24. options = **new** PolylineOptions().addAll(trace).width(8)
25. .color(Color.argb(255,255-255\*counter/11,255\*counter/11,255\*counter/11));
26. }
27. aMap.addPolyline(options);
28. }
29. }

在应用上实时记录并显示轨迹的效果如下图所示。





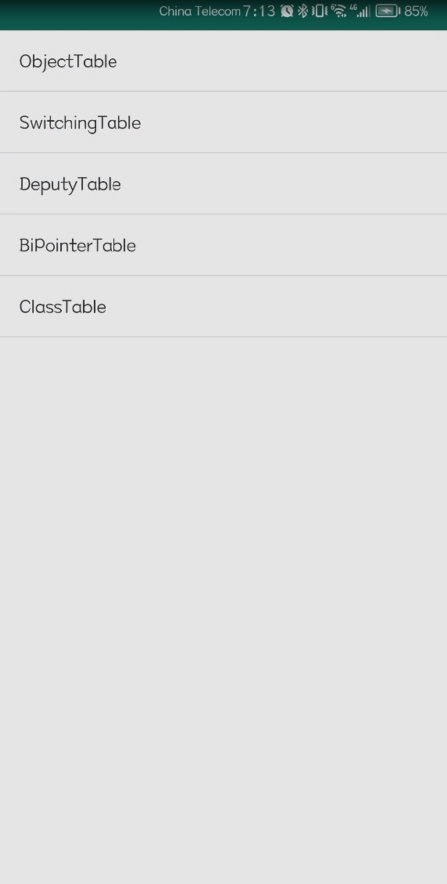
此外，本组开发者在应用界面下方添加了Join Button，便于测试时更加方便地实现路径JOIN操作。为了防止退出应用后数据丢失，本组开发者还修改了MemManager.java文件实现了轨迹数据的持续性存储功能。



**2.3.4 测试与演示**

完整视频存放在在github仓库的report/demos文件夹下，本小节仅以图片展示应用效果。

部分系统表内容如下图所示。



轨迹追踪与可视化效果如下图所示。





轨迹JOIN和可视化效果如下图所示。前者是JOIN前的轨迹；后者是JOIN后的轨迹。





**结论**

在本实验中，本组开发者借助github平台协作完成了三个任务：设计系统表，实现系统表的读取和保存；实现SQL解析中的对象JOIN操作；实现轨迹数据的join和可视化。最后成功基于TOTEM数据库和高德地图API实现了在安卓应用上JOIN用户的行动轨迹并可视化。

通过本次实验，本组成员对数据库系统实现、Android开发等技术有了进一步的领悟。同时，每位成员都有明确分工，通过代码同步和交流使得项目稳步高效推进。本次实验展现了小组的实践能力和合作能力，有效检验了数据库系统理论和实验部分的学习成果。

**参考文献**

1. 珞珈图腾数据库实验室. TOTEM数据[DB/OL]. <http://totemdb.whu.edu.cn/aspx/main/zhanshi_show.aspx?id=1>
2. Android Studio开发者. 探索Android Studio[DB/OL]. <https://developer.android.google.cn/studio/intro?hl=zh-cn>
3. Wikipedia. Android Studio[DB/OL]. <https://en.wikipedia.org/wiki/Android_Studio>
4. 高德开放平台. Android 地图SDK[DB/OL]. <https://lbs.amap.com/api/android-sdk/summary>