Addressing Human Scalability Through Multi-User Editing Using Revision Databases

John Rittenhouse

johnr@ccpgames.com

Overview

- Introduction
- The Problem
- The Solution
- Revisioned Database Overview
- System Implementation Overview
- Examples
- Performance Hotspots
- Issues
- Summary

Who is CCP?

- Independent MMO Company with 600 employees
- Three MMO Projects
  - EVE – Sandbox space ship game with ~370k subscribers, 65k PCU
  - Dust 514 – FPS MMO set in the EVE Universe on the PS3
  - World of Darkness – MMO based on the White Wolf Property
- We need robust game editing solutions

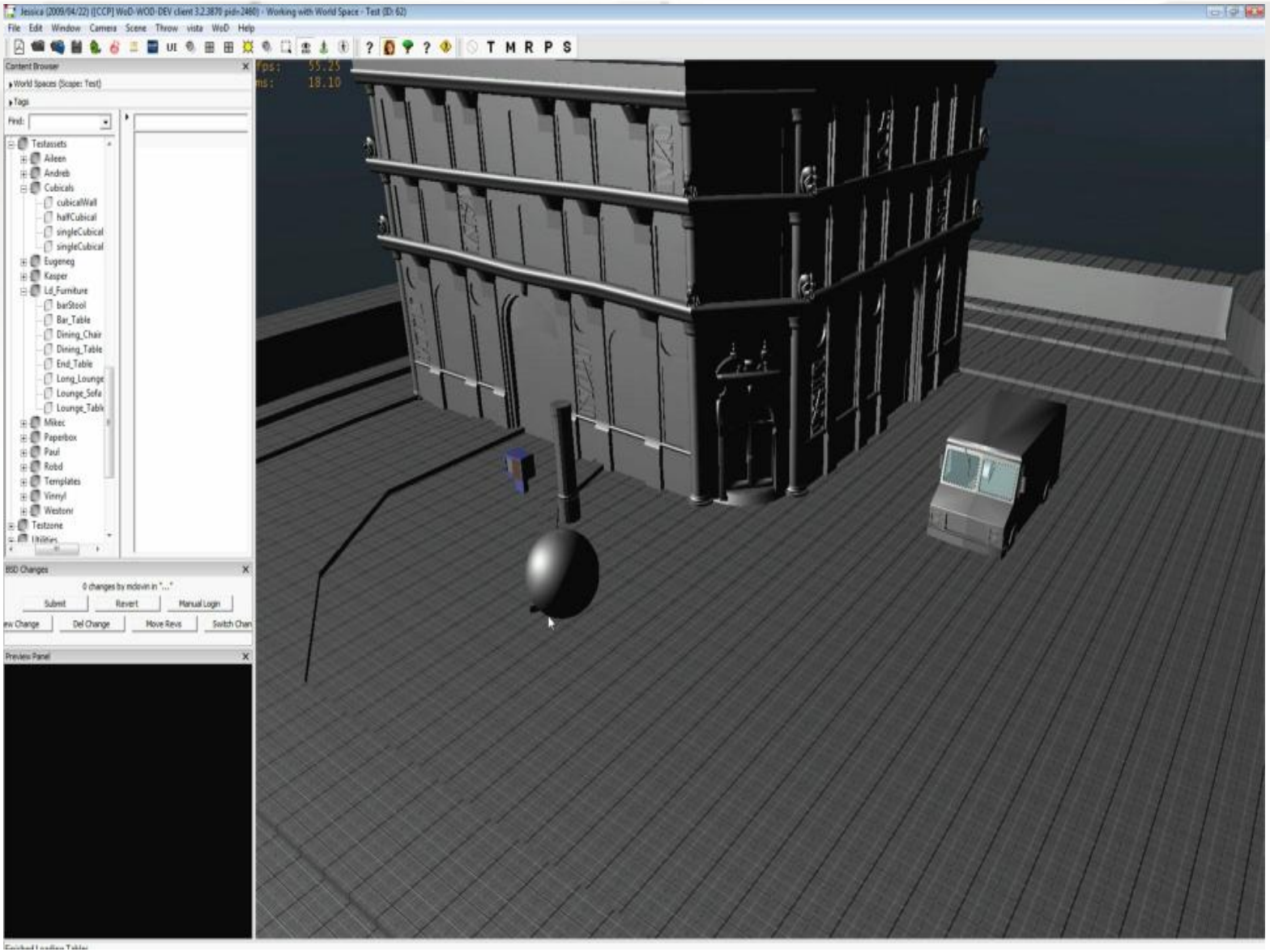
Why address content scalability?

- Customers are seeking larger and more detailed gaming worlds
- Content teams are becoming larger
  - Generalist approach to content creation
    - General level designers focusing on smaller and smaller areas
  - Specialist approach to content creation
    - Level designers, lighters, scripters, etc.
    - Results in user clashing
      - Locked out files
      - Unable to see what others are doing

Possible Solutions

- Mergable File Formats
  - YAML or other text based file formats
  - Issue: Will often require a programmer to help merge
- Layered Levels
  - Different parts of the levels
  - Issue: Often different users will work on the same part and can't see the changes that other users have made but have not submitted or synced to latest
- Realtime Multiuser Editing
  - Changes by users are visible to other users in real time
  - Issue: More complex to implement than other options

Multiple Users Editing Simultaneously



Minecraft Example

Problems to Tackle

- Synchronization of data amongst users
- How to lock the data
- Alerting systems of other user changes

CCP's Problems to Tackle

- Minimal server reboots (Live Editing)
- Multiple users editing the same area
- Backwards compatible with existing database tables
- Support potentially up to 100 content developers
- Ease of use for programmers
  - Transparent
  - Efficient
- System tolerable of high latency
  - Has to handle Trans-Atlantic latency
- Needs to be implemented in Stackless Python

Possible Implementations of MultiUser Editing

- Server Authoritative Editing
  - Server would know immediately when changes occurred
  - Relies on the server always being up for content developers
- Revisioned Database
  - Allows multiple users to see the same set of changes
  - Have to design an easy API to use

What are Revisioned Databases?

- Think of them as version control for databases
- Similar concepts as version control
  - Submitting/Reverting
  - Changelists
  - Locking
- Rows are your atomic unit

Revisioned Database Tables

- Revision Table
  - Tracks all revisions and in which table and key did they occur with
  - Each of them has a changelist id they are in
- Changelist Table
  - Tracks all the changelists and whether they have been submitted
- User Table
  - Who can edit the data
- Recent Changes Table
  - Tracks all recent changes for polling purposes
- Data Tables
  - Table with all the changes

Data Table Example

- Lets say we are labeling fruit
- So our columns for this table will be fruitID and fruitName
- Need also revisionID and changeType



Data Table Example

revisionID	fruitID	fruitName	changeType
0	0	Apple	Add

Table View

revisionID	fruitID	fruitName	changeType
0	0	Apple	Add

Data Table Example

revisionID	fruitID	fruitName	changeType
0	0	Apple	Add
1	0	Fuji Apple	Edit

Table View

revisionID	fruitID	fruitName	changeType
1	0	Fuji Apple	Edit

Data Table Example

revisionID	fruitID	fruitName	changeType
0	0	Apple	Add
1	0	Fuji Apple	Edit
2	1	Grape	Add

Table View

revisionID	fruitID	fruitName	changeType
1	0	Fuji Apple	Edit
2	1	Grape	Add

Data Table Example

revisionID	fruitID	fruitName	changeType
0	0	Apple	Add
1	0	Fuji Apple	Edit
2	1	Grape	Add
3	0	Red Delicious	Edit

Table View

revisionID	fruitID	fruitName	changeType
3	0	Red Delicious	Edit
2	1	Grape	Add

Data Table Example

revisionID	fruitID	fruitName	changeType
0	0	Apple	Add
1	0	Fuji Apple	Edit
2	1	Grape	Add
3	0	Red Delicious	Edit
4	2	Raspberry	Add



Table View

revisionID	fruitID	fruitName	changeType
3	0	Red Delicious	Edit
2	1	Grape	Add
4	2	Raspberry	Add

Data Table Example

revisionID	fruitID	fruitName	changeType
0	0	Apple	Add
1	0	Fuji Apple	Edit
2	1	Grape	Add
3	0	Red Delicious	Edit
4	2	Raspberry	Add
5	1	Grape	Delete

Table View

revisionID	fruitID	fruitName	changeType
3	0	Red Delicious	Edit
4	2	Raspberry	Add

Locking

- Locks occur on a row level
  - Only one user is allowed to change a row at a time
- If a row is not in a submitted change list then it is a locked row.
- Database should reject any changes on locked rows by other users

Syncing

- Copies data from one DB to another DB
- Filtered on
  - Submitted/Unsubmitted Changelists
  - Revision Number
  - Branch

Branching

- Works on the same database instead of trying to merge two databases together
- Change lists can be assigned to a particular branch
- Uses a promotion branch model

Brief Revisioned Database Summary

- Row changes
  - Tracked and stored with revision number and change list
  - Can be submitted or reverted through a change list
  - Rows are locked to other users till their associated change list are submitted or reverted
- Change Lists
- Syncing
- Branching

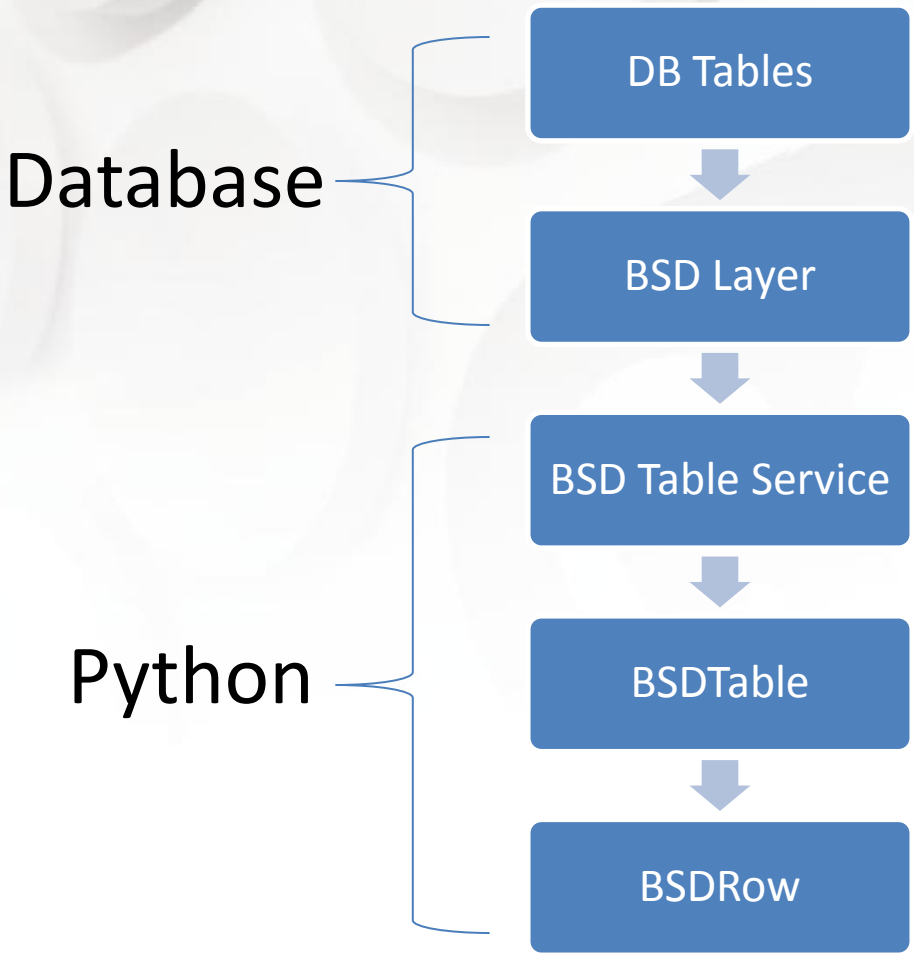
Handling Updates

- Remote Updates - Poll Recent Revisions Table
  - Retrieves table and keys of rows since last check
  - Tables informed to update those rows
- Scatter Update Events on Local/Remote Changes
  - Alerts systems that are listening to the table, rows, and columns that changed with previous and new values
  - Originally just alerted about tables and rows but not the actual data values that were changed (made it hard for systems to minimize processing)

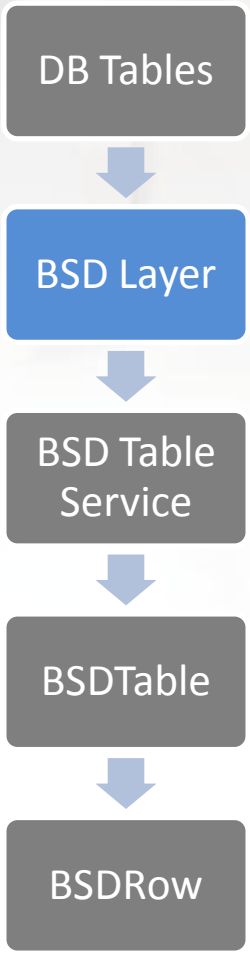
CCP's Revisioned Database System

- Called Branched Static Data (BSD)
  - Developed originally by Jörundur Matthíasson
- Beyond the Database we added layers to Python to ease usage by other programmers

Layer Overview



Branched Static Data (BSD)

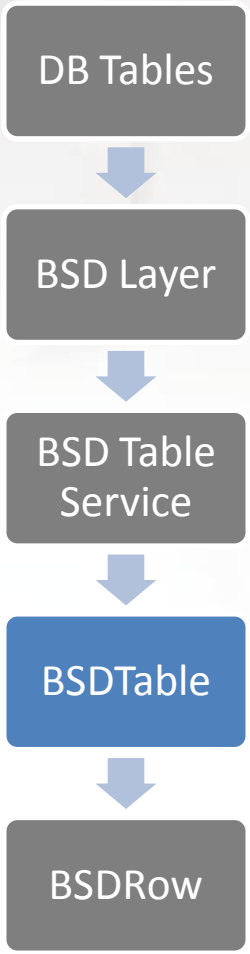


- Internal CCP Revisioned Database
  - Uses views to remain backwards compatible
  - Made of SQL tables, views, and stored procedures
- Per Row Operations
  - Add/Edit/Delete
  - Rows Locked to Single User
- Submit/Revert
- History of Changes
  - Table with all Changes
  - View with most recent
- Branching/Syncing
- Recent Revisions Table

BSD Table Service

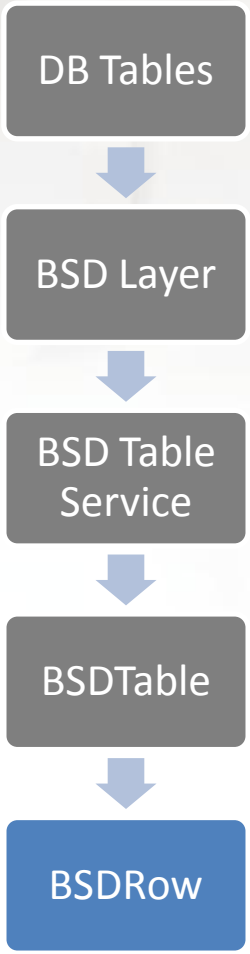
- Holds references to each table
  - Each table is a Python Class
  - GetTable function
- Responsible for Table Updates
  - Polls recent revisions table
  - Alerts tables
  - Handles update tasklets

BSDTable Class



- Loads the Table Data
  - Loads row data into BSDRows
  - Responsible for Indexing
- Holds references to the Rows
  - GetRowByKey
  - GetRows (Filtering)
- Handles Adding/Deleting of Rows
  - AddRow
  - DeleteRow(s)

BSDRow Class



- Handles the data at the row level
- Columns accessed via Properties
  - print row.columnName
  - row.columnName = 2
- Responsible for data editing
  - Threading
  - Merging edits (bucketing)

Example – Table Definition

Schema Table Name Label Key ID 1 Key ID 2

```
--DBTOOLS BSDTABLE 'world', 'staticWorldSpaceObjects', 'objectName', 'worldSpaceID', 'objectID'
CREATE TABLE world.staticWorldSpaceObjects
(
  worldSpaceID          int             NOT NULL,
  objectID              int             NOT NULL,

  objectName            nvarchar(64)    NULL      DEFAULT 'Unnamed',

  -- Position
  posX                  float           NULL      DEFAULT 0.0,
  posY                  float           NULL      DEFAULT 0.0,
  posZ                  float           NULL      DEFAULT 0.0,

  -- Rotation (Computed in the order Yaw, Pitch, Roll)
  rotX                  float           NULL      DEFAULT 0.0, -- Pitch
  rotY                  float           NULL      DEFAULT 0.0, -- Yaw
  rotZ                  float           NULL      DEFAULT 0.0, -- Roll

  -- Scale
  scaleX                float           NULL      DEFAULT 1.0,
  scaleY                float           NULL      DEFAULT 1.0,
  scaleZ                float           NULL      DEFAULT 1.0,

  graphicID             int             NULL,
)
GO
```

Example – Python Code

```
# Get the BSDTable service
bsdTable = sm.GetService("bsdTable")

# Get the world space objects table
wso = bsdTable.GetTable("world.staticWorldSpaceObjects")

# Get an object with worldSpaceID 62 and objectID 8 then set it to the origin
obj = wso.GetRowByKey(62, 8)

print "Setting '%s' to the origin"%obj.objectName
obj.posX = 0
obj.posY = 0
obj.posZ = 0

# Count how many objects exist the objectName "chair_wood_windsorRes"
objs = wso.GetRows(objectName = "chair_wood_windsorRes")
print "Found", len(objs), "objects with the name 'chair_wood_windsorRes'"

# Create a new object in worldSpaceID 8 then delete it
revisionID, worldSpaceID, objectID = wso.AddRow(8, objectName="Object to be deleted")
print "Added an object to worldSpaceID", worldSpaceID, "with objectID", objectID, ", but deleting it now"
wso.Delet
```

Output

Setting 'Ball' to the origin
Found 302 objects with the name 'chair_wood_windsorRes'
Added an object to worldSpaceID 8 with objectID 33 , but deleting it now

Performance Hotspot - Filtering

- Indexed on Columns
  - Initially indexed into lists but swapped to sets
    - Relational databases uses set theory to be fast so lets do the same
  - Choosing columns to index on
- Used to use the DB to filter if our data isn't fully loaded
  - Resulted in a short term boost for a longer
- Future Changes
  - Case insensitive & delete/nondeleted Indexing
  - Caching/filtering using SQLite

Performance Hotspot - Transactions

- Operations often involve multiple rows
  - Adds often depend on the keys of previous adds
  - Occurs when there is a main table plus additional optionable tables
  - DB Latency makes waiting on responses to slow
- Allows merging of all BSD operations in a tasklet into a single DB query
- Provides traditional benefits of entire operations written at once to DB
- Easy to Use
  - TransactionStart()/TransactionEnd(transactionName)
  - with bsd.Transaction(transactionName):

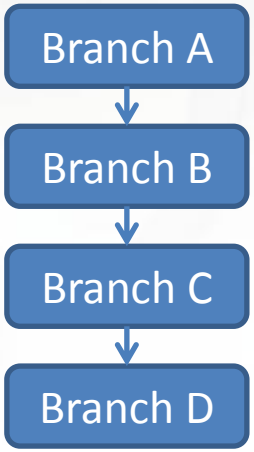
Common Pitfalls for Programmers

- Writing Cacheless Algorithms
  - Relies on BSD Table Services for caches
  - Makes creating live systems easier
  - Often results in higher order algorithms
- Assuming Data is Loaded Transaction Based
  - Data written as a transaction isn't guaranteed to be loaded in a single operation
  - Possible Solutions
    - Allow checking to see if a full Recent Revision Update is completed
    - Actually Setup Loading to be transaction based

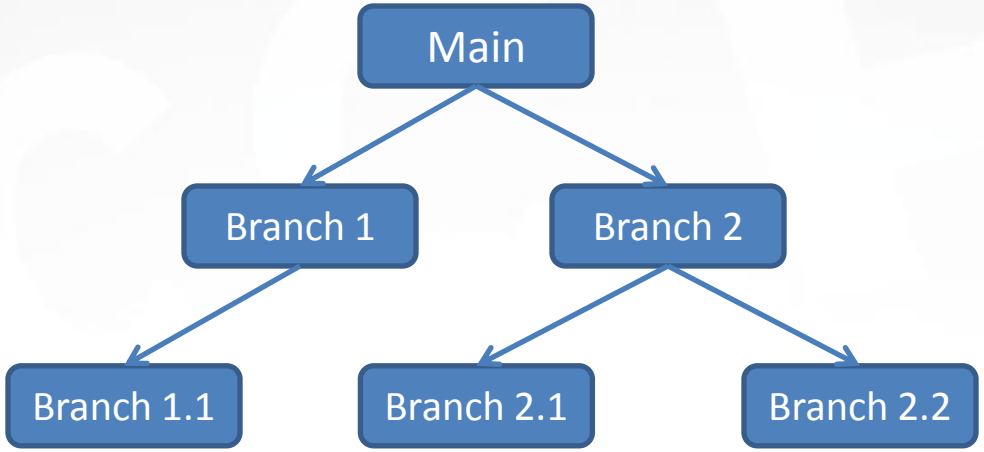
Issues – Promotion Branching

- We have moved from a promotion to mainline branch model

Promotion Branching



Mainline Branching



Issues – Promotion Branching

- Causes issues with non-backwards compatible changes necessary in other branches
  - Handled with scripts to move the static data between branches
- Interlinked data needs to be set to the same branch
  - Hard to determine the issues without validation
  - Weird Testing flag which makes the issue even worse

Replacing with a Mainline Branch Model

- Databases
  - Main Database
    - Responsible for all table ID's
  - All other DB's derive from the Main DB
  - Databases will correspond to the equivalent Perforce Branch
- Code and Databases will be integrated together
  - Databases track the change list number they have been integrated from and to
  - Table merges occur automatically except for row conflicts
  - Conflicts at the row resolved by choosing one side or the other
  - Does not handle all potential issues

Summary

- Revisioned Databases work well for multiuser editing
  - Locking is often handled automatically within them
  - Easy to determine changes
- Ease of use is important for programmers
- For a large project assume the programmer won't know what you consider to be the "normal" use case
  - Optimize based on use cases (programmers learn by example)
- DB branching model needs to match the code branching

CCP Is Hiring

Apply online at http://www.ccpgames.com/en/jobs.aspx

Jobs available in

Atlanta, USA

Reykjavik, Iceland

Shanghai, China