# Serving Database Information Using a Flexible Server in a Three Tier Architecture

## DAN

(Database Architecture Networked)

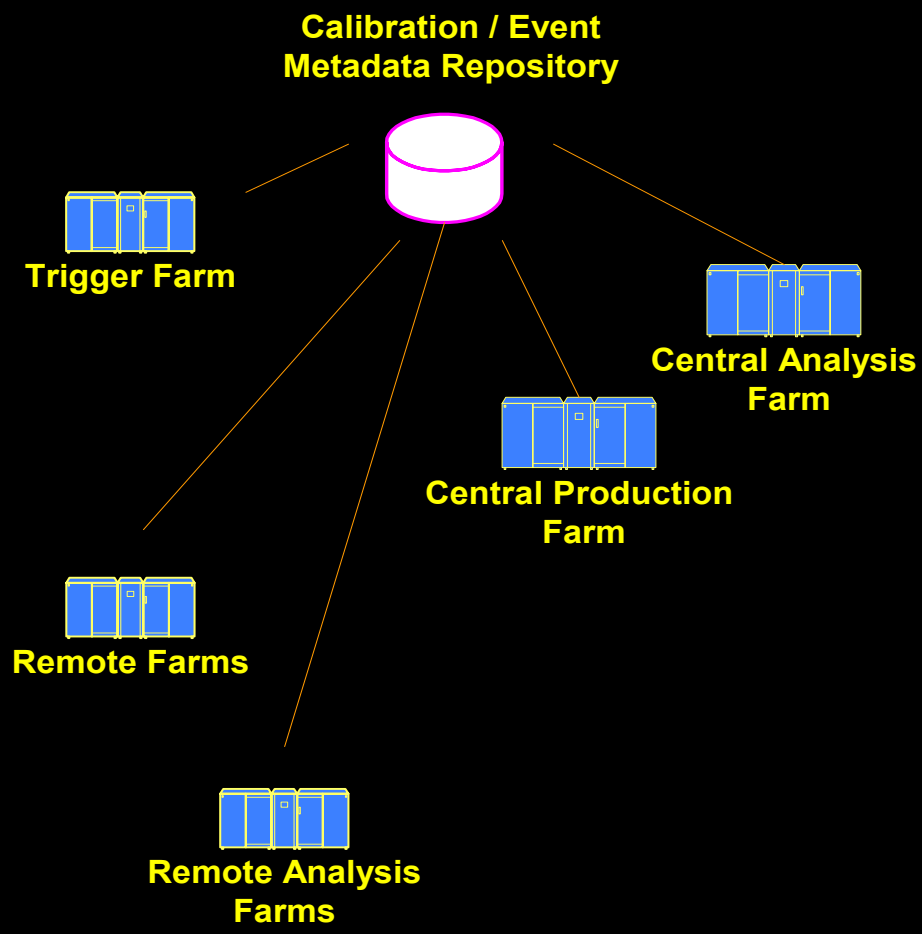
Presented by Jim Kowalkowski
Fermilab

# Project History

- Conceived and prototyped as part of the low-level services of the Calibration and SAM database server projects for DØ
- DAN is a second generation system
- Development Team
  - Stephen White, Herbert Greenlee, Robert Illingworth, Jim Kowalkowski, Anil Kumar, Margherita Vittone, Taka Yasuda
  - Lee Lueking, Vicky White (requirements)

# The Problem

- Supporting thousands of repository accesses per hour for constants, configuration, and dataset information
- Delivering data to all the jobs in a timely fashion

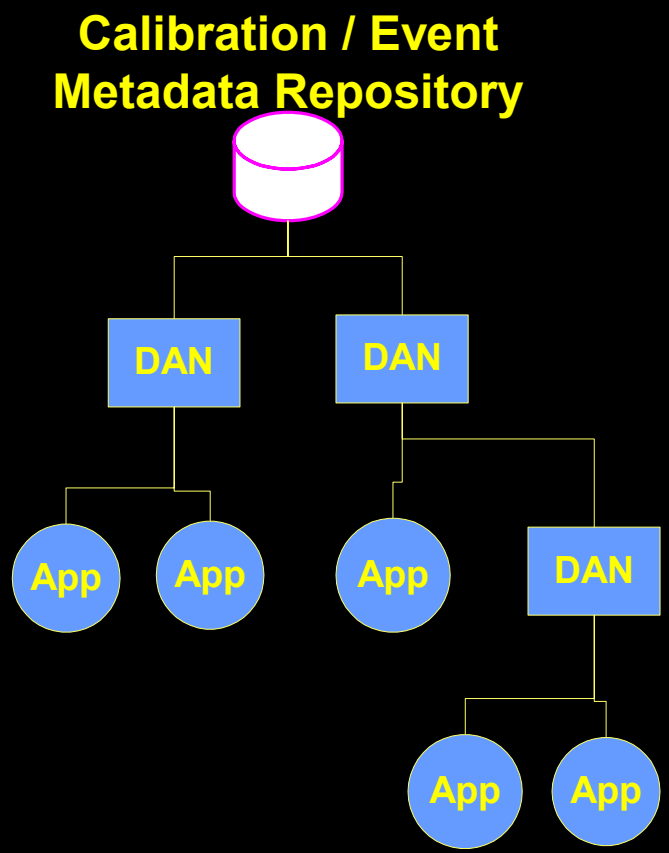


# Requirements

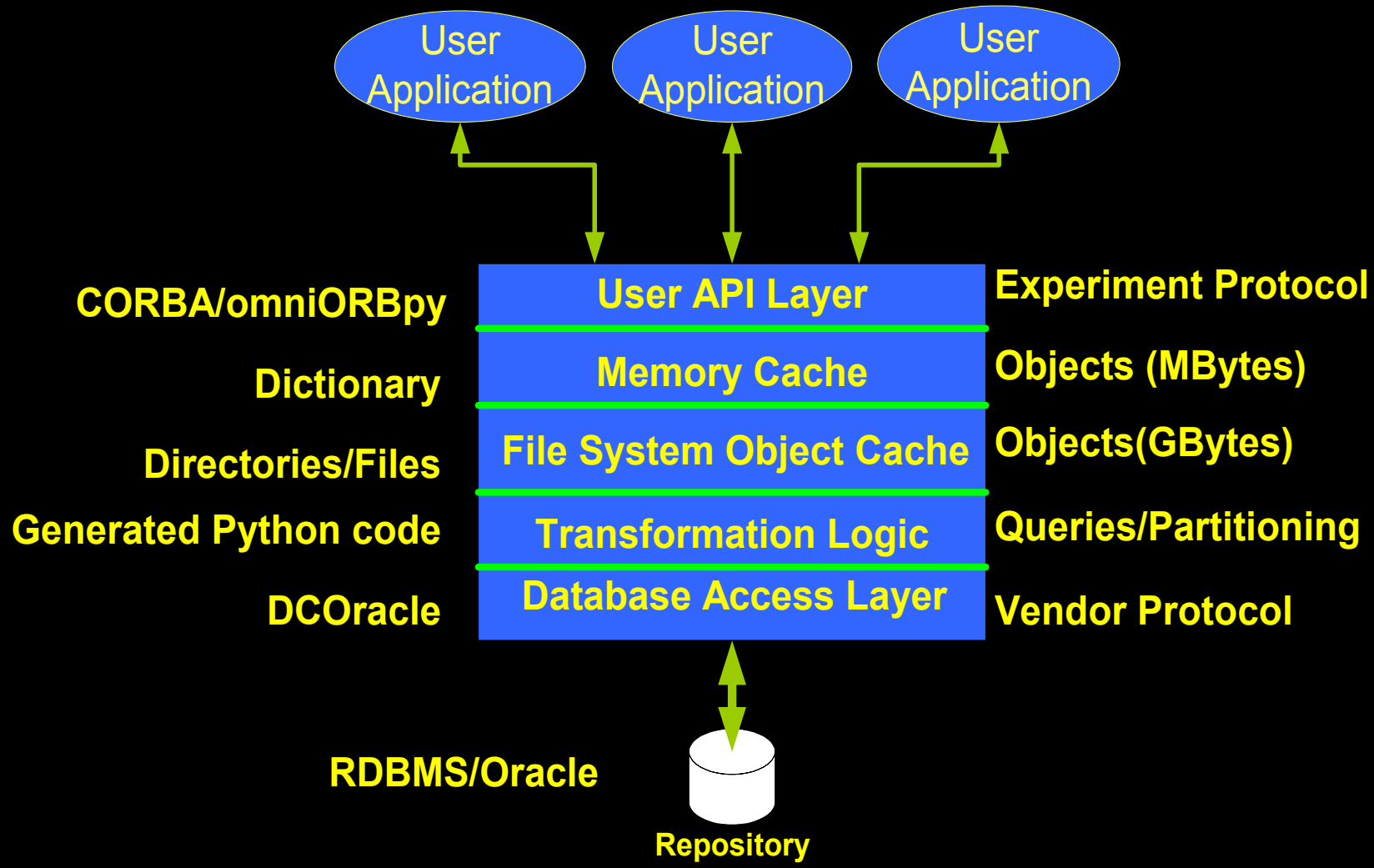
- Applications not tied to a database vendor and API
- Low latency for multiple simultaneous client requests
- Controlled access to database
- Centralized queries and knowledge of schema
- Remote servers operational during network and database outages
- Scales in a controlled fashion
- Easily tuned, configured, administered
- Detailed activity monitoring and error handling
- Answer Calibration and event metadata queries

# What is DAN?

- A multi-tiered Python server between the database and the user applications
- A server that performs database transactions on behalf of the user
- A service that provides an application-level protocol for accessing calibrations and event meta data



# Architecture and Design



# Features

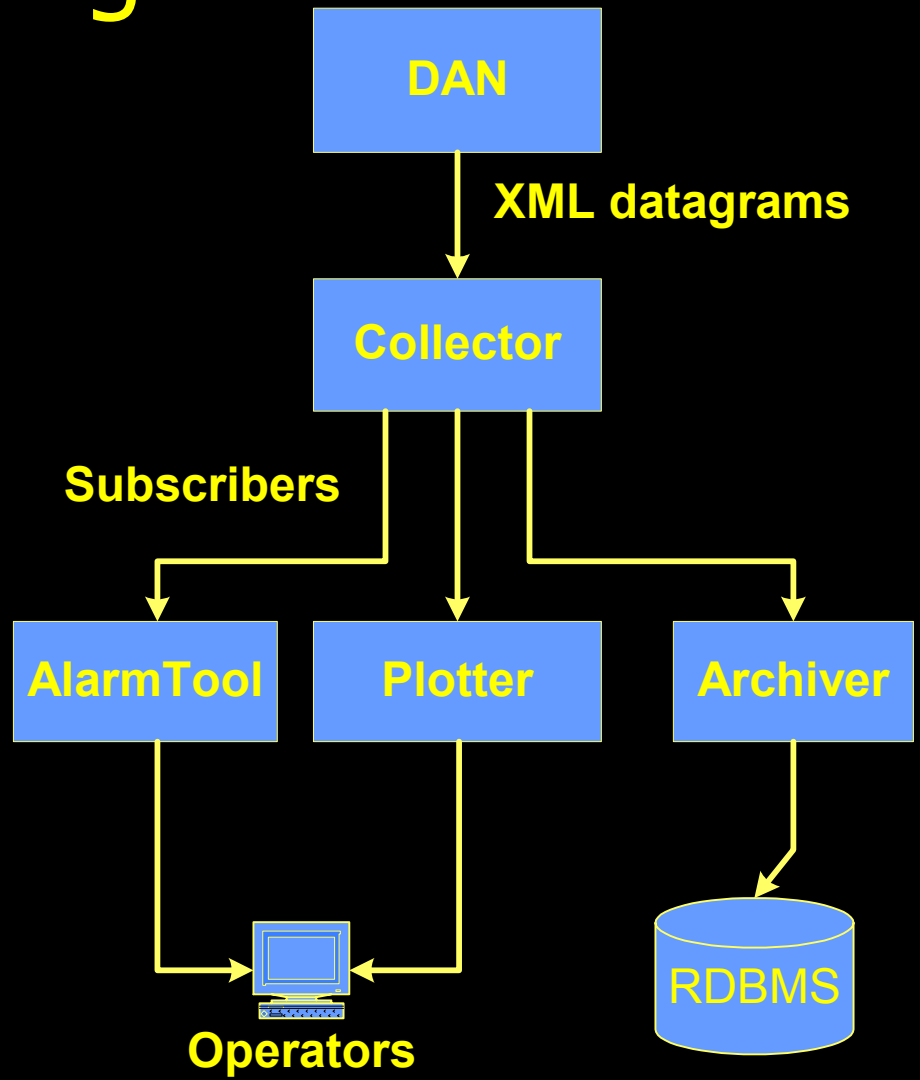
- Multi-level cache strategy
  - Fast delivery of commonly used objects
  - Reduces transactions to database server
  - Avoid coherency problems by using “WORM” data
- Database connection pooling
  - Consolidate many of the same transaction
  - Reduce the number of concurrent users
- Code generation
  - Reduce maintenance
  - Capture common usage patterns
  - Reduce development time

# Features(2)

- Table to object transformation policies
  - Calibrations delivered in chunks by crate
  - Complex joins to form single objects
- Load balancing
  - Transaction throttling
  - Information source selection
  - Proxy mode supported
- Multi-threading
  - Reduce latency
  - Serve cached objects while queries are taking place

# Statistics Gathering

- Monitor resource usage and activity through subscription service.
- Verbose recording of interesting events (errors, informational, or debugging).
- Threshold based asynchronous notifications.
- Problem solving aid



# Testing and Evaluation

- *Very difficult to schedule adequate resources for doing good load testing!*
- 800,000 rows calibration dataset (~25MB) sent to 50 farm nodes test
  - 6.5 minutes to complete
  - 3 – 4 times faster to hit cache than database
  - ~300MB memory per dataset in cache
  - <30% CPU usage for DAN and database nodes (dual Athlon 1800)
  - ~3.2MB/s delivery rate

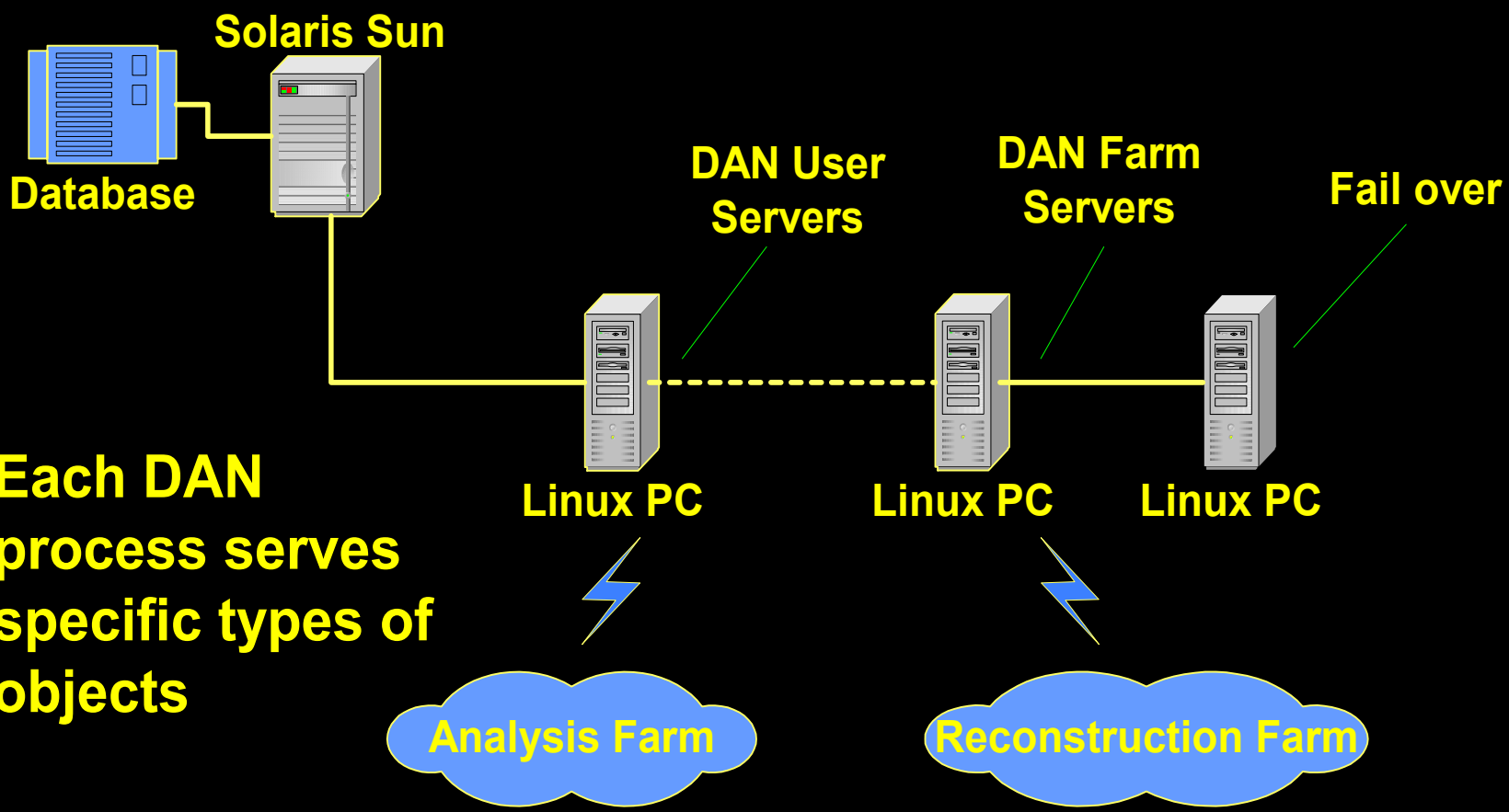
# Operational experiences

- Third party products
  - DCOracle (Python API to Oracle)
    - poor backward compatibility between versions
  - omniORBpy (CORBA implementation)
    - Random thread lockup problems
    - Very difficult to locate problem sources
- ~30% CPU load in production on a dual Athlon 1800 node with nine DAN servers running
- Configuring a remote server requires "expert" input due to complexity of choices
- Large amount of RAM required for caching a complete silicon vertex dataset (~300MB)

# Were good choices made?

- CORBA
  - Access to calibration data not very object oriented
  - CORBA data representations are not useful in application
  - Better suited for event metadata queries
- Python choice
  - Multi-threading limited to one CPU
  - Good choice for scripting language
    - Simplified 3rd party product integration
    - Faster development and deployment
  - Not as good a choice for production
    - High memory requirements
    - Poor debugging tools
    - Slower execution speed than other language choices
- Code generation
  - Single edit point for code changes to multiple classes
  - Difficult to understand how it works

# Deployment




**Each DAN process serves specific types of objects**

# Future plans

- Alternative "experiment protocol"
  - Client using HTTP with cURL
  - Server using Apache with modpython
  - XML datagrams
- Completion of the statistics gathering, remote monitoring, controls package, and graphs available via the web
- Changing resource allocations or other configuration parameters dynamically
- Addressing the memory cache RAM requirements

# End

# Extra Information…

- Effort to produce was 1 fte for 1.5 years
- Expected effort required to maintain is .1 fte per year
- Transaction rate varies from 300 to 3000 requests per hour
- Transaction size varies from 1.8kb to 4.5kb. (for the tested server)
- Number of active servers we expect
  - 6 – 10 high demand centers each running 8 servers
  - 80 institutions whose server configuration is variable
- Amount of memory and CPU required (for largest server)
  - 300mb of RAM per data set
  - A server uses under 30% of the CPU when under load (1533mhz dual processor)
  - Database server uses 30% of the CPU to query and return a 25mb data set.